

国际信息工程先进技术译丛

WILEY

计算机系统设计：片上系统

Computer System Design: System-on-Chip

[美] 迈克尔 J.弗林（Michael J.Flynn）
[英] 陆永青（Wayne Luk） 著
张志敏 范东睿 等译



机械工业出版社
CHINA MACHINE PRESS

国际信息工程先进技术译丛

计算机系统设计：片上系统

[美] 迈克尔 J. 弗林（Michael J. Flynn）
[英] 陆永青（Wayne Luk） 著
张志敏 范东睿 等译



机 械 工 业 出 版 社

图书在版编目（CIP）数据

计算机系统设计：片上系统/（美）弗林（Flynn, M. J.），（英）陆永青著；张志敏等译. —北京：机械工业出版社，2015.4
（国际信息工程先进技术译丛）
书名原文：Computer system design: system-on-chip
ISBN 978-7-111-49813-1

Ⅰ. ①计… Ⅱ. ①弗…②陆…③张… Ⅲ. ①电子计算机—系统设计 Ⅳ. ①TP302.1

中国版本图书馆 CIP 数据核字（2015）第 062738 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
策划编辑：刘星宁 责任编辑：刘星宁
责任校对：樊钟英 封面设计：马精明
责任印制：刘 岚
北京中兴印刷有限公司印刷
2015 年 6 月第 1 版第 1 次印刷
169mm×239mm · 18.5 印张 · 358 千字
0 001—2 500 册
标准书号：ISBN 978-7-111-49813-1
定价：78.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务	网络服务
服务咨询热线：010－88361066	机 工 官 网：www.cmpbook.com
读者购书热线：010－68326294	机 工 官 博：weibo.com/cmp1952
010－88379203	金 书 网：www.golden-book.com
封面无防伪标均为盗版	教育服务网：www.cmpedu.com

本书由计算机工程领域资深学者编著，涵盖了计算机系统/SoC设计的许多重要研究内容，着眼于以系统为中心的设计空间理念，从基本概念和分析技术着手，对各种应用和架构设计、开发予以重点阐述。书中除了讲解计算机体系结构中处理器、内存、互联等要素外，重点介绍了系统的定制化设计技术与可重构性设计技术，更关注系统级开发时关于面积、速度、功耗和可配置性等权衡技术发展，并指出计算机系统/SoC 设计面临的挑战。

本书不仅可供计算机系统设计专业人员、SoC 设计师及计算机学者阅读，也可作为计算机科学、计算机工程及电子工程等专业研究生的参考书。

译　者　序

随着计算机与微电子技术融合趋势日益凸显，以及 CPU/SoC 设计技术快速发展，计算机体系结构课程的挑战性越来越明显，需要专业人员深入研究微体系结构，也给培养创新型计算机人才带来了极大的挑战和机遇。

早在几年前，拜读《Computer System Design：System-On-Chip》，就引起我们共鸣。我们翻译组主译人员从事 SoC 研究设计开发近 20 年。“2002 年中国未来十大技术之一——SoC 技术”[⊖] 就由主要译者张志敏点评，并且他一直带领团队不断研究开发一系列 SoC，培养了一批芯片设计人才。我们始终认为 SoC 能够为计算机学科发展注入新鲜活力，必将丰富计算机体系结构教学课件内容，利于造就一大批优秀的计算机系统研究开发人才。

原书作者 Michael J. Flynn 是美国哈佛大学教授，1966 年提出弗林分类学，1995 年获哈利·古德纪念奖，2009 年获贝尔格莱德大学荣誉博士。作者多年来一直从事计算系统工程设计和体系结构教学研究工作，设计经验丰富。他编著的原书英文版内容翔实，涵盖了计算机系统设计的许多方面，从基本概念和分析技术着手，就各种应用和架构的设计开发予以重点阐述，对于计算机专业师生和系统集成芯片设计开发者而言，是一本很好的参考书。

本书的翻译工作由范东睿组织完成，宋风龙和孟海波协助执行，参与本书前期辅助翻译工作的有廖飞、马丽娜、谭旭、李文明、王宏博、申小伟、郑亚松，张志敏参与较多的翻译工作，并负责统稿、校对、润色等工作。

感谢机械工业出版社给我们翻译这样一本好书的机会，感谢编辑为本书出版所做的工作。

译者

2015 年 4 月

⊖ 由《计算机世界》报业集团主办。

原书前言

计算机系统设计者下一步将更关注系统定制元素，以针对特定的应用，而非处理器和存储器系统的细节。这样的设计者应具有处理器和其他部件的基本知识，但他们设计成功与否将取决于他们对系统的平衡能力，以及在可优化成本、性能和其他满足应用需求属性方面的能力。本书将介绍计算机系统设计，特别是SoC设计中的问题。

驾驭这样的设计需要一系列知识，如图0.1所示。本书第1章对系统方法进行了简介绪论，第2章着眼于定义设计空间——面积、速度、功耗和可配置性，第3~5章提供系统的基本元素的背景知识——处理器、内存和互联。后续的章节专注于面向特定应用程序和技术的计算机系统定制：第6章涵盖了定制和配置的设计问题，第7章讨论了针对各种应用的系统级平衡技术，将早期的材料归类在一起研究，第8章提出系统设计和SoC设计未来可能面对的挑战。本书所描述的工具仍在发展中，附录提供了一些工具概述。由于工具不断革新，请经常看看公司的网站：www.soctextbook.com。此外，对教学有用的材料，如幻灯片和练习的答案，也在准备中。

本书涵盖了计算机系统设计的一个特定的方法，介绍的基本概念和分析技术适用于各种应用和架构，而不是针对特定的应用、架构、语言和工具。本书还包括互补处置和其他主题，如电子系统级设计、嵌入式软件开发和系统级集成和测试。在适当的地方，本书简短地描述和引用这些主题，更详细的处置可能包含在未来的版本中或其他书中。

SoC是一个快速发展的领域。虽然专注于基本的资料，但为完成本书也简要介绍了最新的技术进展。当然，这些最新的相关技术，可到如网站等相关的信息源找寻。

许多同事和学生，多数来自英国伦敦帝国理工学院和美国斯坦福大学，对本书做出了贡献。很抱歉，难以在这里说出他们所有人的名字。然而，一些人应该特别感谢，Peter Cheung从一开始就与我们合作，他的贡献体现在许多主题上，特别是第5章；Tobias Becker、Ray Cheung、Rob Dimond、Scott Guo、Shay Ping Seng、David Thomas、Steve Wilton、Alice Yu和Chi Wai Yu等对各章节提供了重要素材；Philip Leong和Roger Woods多次精读了手稿并提供了许多改进建议；也从Jeffrey Arnold、Peter Boehm、Don Bouldin、Geoffrey Brown、Patrick Hung、Sebastian Lopez、Oskar Mencer、Kevin Rudd和匿名审稿人等获益良多；感谢Kubi-

lay Atasu、Peter Collingbourne、James Huggett、Qiwei Jin、Adrien Le Masle、Pete Sedcole 和 Tim Todman 等提供的宝贵援助。

最后，感谢美国 Wiley 出版社的 Cassie Strickland、美国 Toppan Best-set 出版社的 Janet Hronek 对及时完成本书提供的帮助。

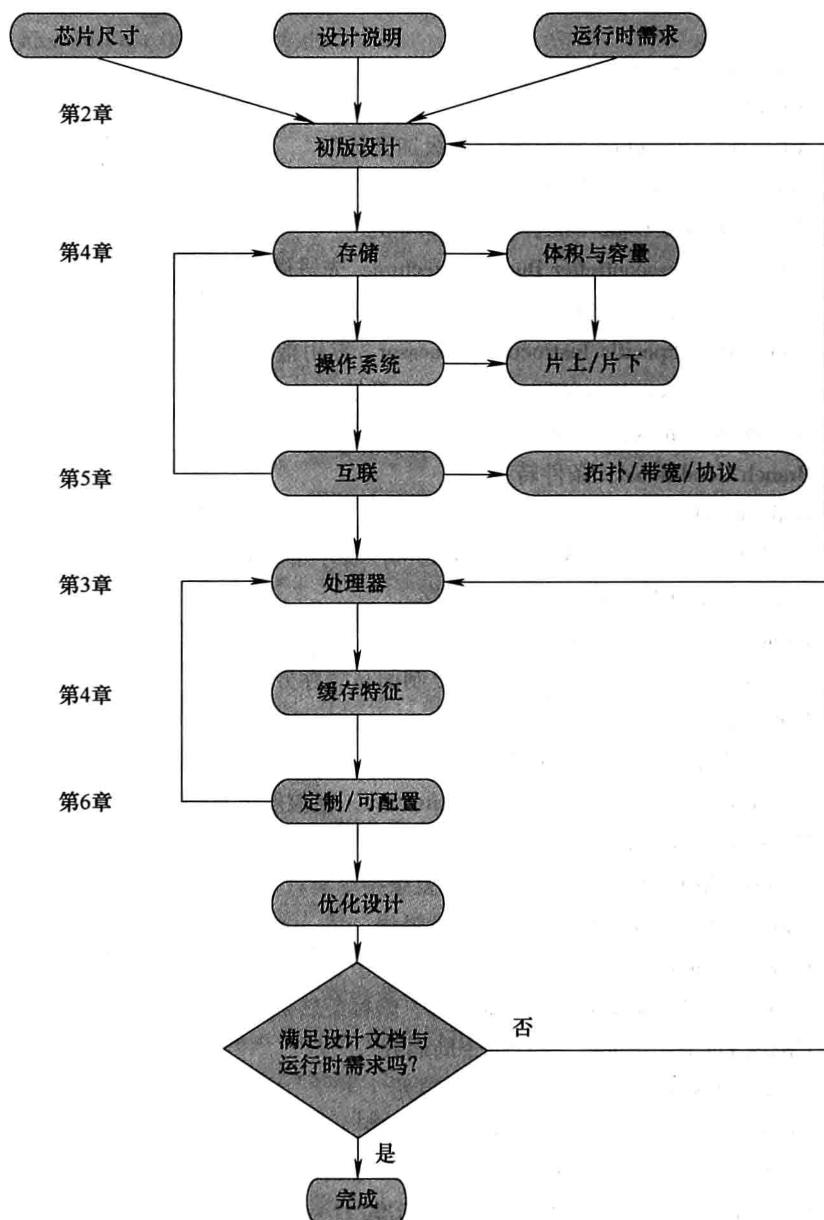


图 0.1　本书所述 SoC 设计方案

缩 略 语

AC Autonomous Chip，自主芯片
A-D Analog to Digital，模-数转换
AES Advanced Encryption Standard，高级加密标准
AG Address Generation，地址形成
ALU Arithmetic and Logic Unit，算术逻辑单元
AMBA Advanced Microcontroller Bus Architecture，先进微控制器总线架构
ASIC Application -Specific Integrated Circuit，专用集成电路
ASIP Application -Specific Instruction Processor，专用指令处理器
ASoC Autonomous System-on-Chip，自主片上系统
AXI Advanced eXtensible Interface，先进的可扩展接口
BC Branch Conditional，条件转移
BIST Built-In-Self-Test，内嵌自测试
BRAM Block Random Access Memory，块随机存取存储器
BTB Branch Target Buffer，分支目标缓存
CAD Computer Aided Design，计算机辅助设计
CBWA Copy-Back Write Allocate cache，复制回写缓存分配
CC Condition Codes，条件码
CFA Color Filter Array，彩色滤波阵列
CGRA Coarse-Grained Reconfigurable Architecture，粗粒度可重构体系结构
CIF Common Intermediate Format，通用中间格式
CISC Complex Instruction Set Computer，复杂指令集计算机
CLB Configurable Logic Block，可配置逻辑块
CMOS Complementary Metal Oxide Semiconductor，互补金属氧化物半导体
CORDIC COordinate Rotation Digital Computer，坐标旋转数字计算机
CPI Cycles Per Instruction，平均指令周期数
CPU Central Processing Unit，中央处理单元
DCT Discrete Cosine Transform，离散余弦变换
DDR Double Data Rate，双数据速率
DES Data Encryption Standard，数据加密标准
3DES Triple Data Encryption Standard，三重数据加密标准
DF Data Fetch，取数据
DMA Direct Memory Access，直接存储访问
DRAM Dynamic Random Access Memory，动态随机存取存储器

DSP	Digital Signal Processing（Processor），数字信号处理（器）
DTMR	Design Target Miss Rates，设计目标的命中率
ECC	Error Correcting Code，纠错码
eDRAM	embedded Dynamic RandomAccess Memory，嵌入式动态随机存取存储器
EX	Execute，执行
FIFO	First In First Out，先进先出
FIR	Finite Impulse Response，有限冲激响应
FO4	Fan-Out of four，4 扇出
FP	Floating-Point，浮点
FPGA	Field Programmable Gate Array，现场可编程门阵列
FPR	Floating-Point Register，浮点寄存器
FPU	Floating-point unit，浮点部件
GB	Giga Bytes，a billion（10^9）bytes，吉字节
GIF	Graphics InterFace，图形界面
GPP	General-Purpose Processor，通用处理器
GPR	General-Purpose Register，通用寄存器
GPS	Global Positioning System，全球定位系统
GSM	Global System for Mobile communications，全球移动通信系统
HDTV	High Definition TeleVision，高清晰度电视
HPC	High Performance Computing，高性能计算机
IC	Integrated Circuit，集成电路
ICU	Interconnect interface Unit，互联接口单元
ID	Instruction Decode，指令译码
IF	Instruction fetch，取指
ILP	Instruction-Level Parallelism，指令级并行
I/O	Input/Output，输入/输出
IP	Intellectual Property，知识产权
IR	Instruction Register，指令寄存器
ISA	Instruction Set Architecture，指令集体系结构
ISEF	Instruction Set Extension Fabric，指令集扩展架构
JPEG	Joint Photographic Experts Group（image compression standard），联合图像专家小组
Kb	Kilo bits，one thousand（10^3）bits，千（1024）位
KB	Kilo Bytes，one thousand bytes，千（1024）字节
L1	Level 1（for cache），一级缓存
L2	Level 2（for cache），二级缓存
LE	Logic Element，逻辑器件
LRU	Least Recently Used，最近少使用的
L/S	Load/Store，取存

LSI	Large Scale Integration，大规模集成
LUT	LookUp Table，查找表格
Mb	Mega bits，one million（10^6）bits，兆位
MB	Mega Bytes，one million bytes，兆字节
MEMS	Micro Electro Mechanical System，微机电系统
MIMD	Multiple Instruction streams，Multiple Data streams，多指令多数据流
MIPS	Million Instructions Per Second，每秒百万指令
MOPS	Million Operations Per second，每秒百万操作
MOS	Metal Oxide Semiconductor，金属氧化物半导体
MPEG	Motion Picture Experts Group（video compression standard），动态图像专家组
MTBF	Mean Time Between Faults，平均故障间隔时间
MUX	MUltipleXor，多路转接器
NOC	Network On Chip，片上网络
OCP	Open Core Protocol，开放核心协议
OFDM	Orthogonal Frequency-Division Multiplexing，正交频分复用
PAN	Personal Area Network，个人区域网络
PCB	Printed Circuit Board，印制电路板
PLCC	Plastic Leaded Chip Carrier，塑料封装芯片载体
PROM	Programmable Read Only Memory，只读存储器
QCIF	Quarter Common Intermediate Format，四分之一通用中间格式
RAM	Random Access Memory，随机访问存储处理器
RAND	Random，随机
RAW	Read-After-Write，写后读
rbe	register bit equivalent，寄存器位等效
RF	Radio Frequency，无线射频
RFID	Radio Frequency Identification，射频识别
RISC	Reduced Instruction Set Computer，精简指令集计算机
R/M	Register/Memory，寄存器/存储器
ROM	Read Only Memory，只读存储器
RTL	Register Transfer Language，寄存器传送语言
SAD	Sum of the Absolute Differences，绝对差异的总和
SDRAM	Synchronous Dynamic Random Access Memory，同步动态随机存取存储器
SECDED	Single Error Correction，Double Error Detection，单纠错双纠错
SER	Soft Error Rate，软件错误率
SIA	Semiconductor Industry Association，（美国）半导体行业协会
SIMD	Single Instruction stream，Multiple Data streams，单指令多数据流
SMT	Simultaneous MultiThreading，并发多线程
SoC	System on Chip，片上系统

SRAM　Static Random Access Memory，静态随机访问存储器
TLB　Translation Look - Aside Buffer，旁路转换缓冲器
TMR　Triple Modular Redundancy，三重模块冗余度
UART　Universal Asynchronous Receiver/Transmitter，通用异步接收器/发送器
UMTS　Universal Mobile Telecommunications System，通用移动通信系统
UV　UltraViolet，紫外光
VCI　Virtual Component Interface，虚拟组件接口
VLIW　Very Long Instruction Word，超长指令字
VLSI　Very Large Scale Integration，超大规模集成
VPU　Vector Processing Unit，向量处理单元
VR　Vector Register，向量寄存器
VSIA　Virtual Socket Interface Alliance，虚拟接口联盟
WAR　Write After Read，读后写
WAW　Write After Write，写后写
WB　Write Back，回写
WTNWA　Write-Through cache No Write Allocate，直写无须写分配

目　　录

第1章　系统方法简介

1.1　系统架构：概览

在过去40年中，大家已经看到了随着硅技术的巨大进步，晶体管的密度与性能有了极大提升。1966年，美国仙童（Fairchild）半导体公司[84]推出了一种四路双输入与非门，它是在芯片上集成了大约10个晶体管。而2008年，美国英特尔公司四核安腾（Itanium）处理器上有20亿个晶体管[226]。图1.1和图1.2给出了在提高晶体管密度方面不断的进步及设备成本方面相应的减少。

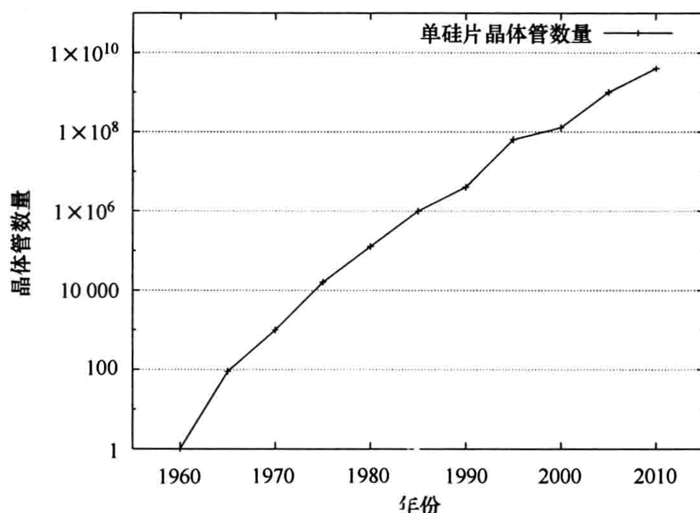


图1.1　不断增长的硅片晶体管密度

本书的目标在于提出一种通过利用这种超高的晶体管密度来进行计算机系统设计的方法。在某种程度上，本书是计算机体系结构和设计研究的直接延伸，当然，它同样也是系统架构和设计方面的研究材料。

大约50年前，一篇具有开创性的文章《系统工程——大规模系统设计导论》[111]问世。其作者H. H. Goode和R. E. Machol指出，系统的工程性观点是由处理复杂问题的需求所产生的。同样，基于计算机的工具极大地增强了我们处理复杂设计问题的能力。

一个片上系统（System-on-Chip，SoC）架构是由处理器、存储器及针对一个

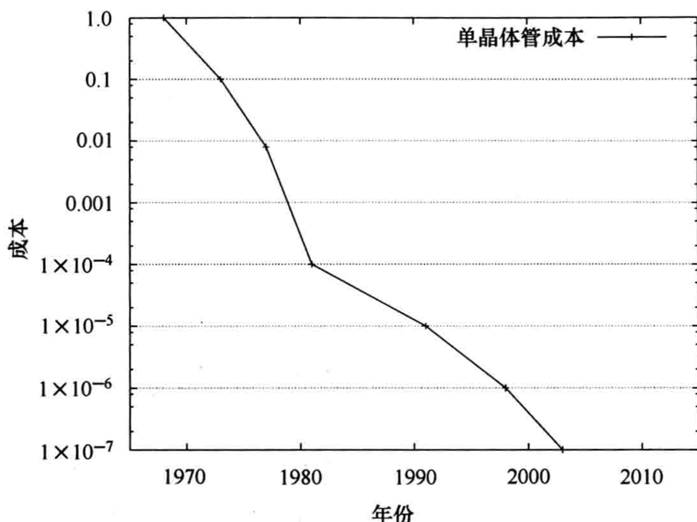


图 1.2 几十年来晶体管成本不断下降

应用领域的互联所构成。日本索尼公司的 PlayStation 2（见图 1.3）的情感引擎（Emotion Engine）[147,187,237]就是一个这种架构的简单例子。它主要有两个功能：行为模拟与几何转换。此系统包含三个最基本的组成部分：一个精简指令集计算机（Reduced Instruction Set Computer，RISC）式[118]的主处理器；两个由 4 个并行单指令多数据流（Single Instruction Multiple Data，SIMD）式[97]处理器组成的向量处理部件（Vector Processing Unit，VPU），VPU0 和 VPU1。在接下来几节将对这些组件及使用方法做简要的介绍。

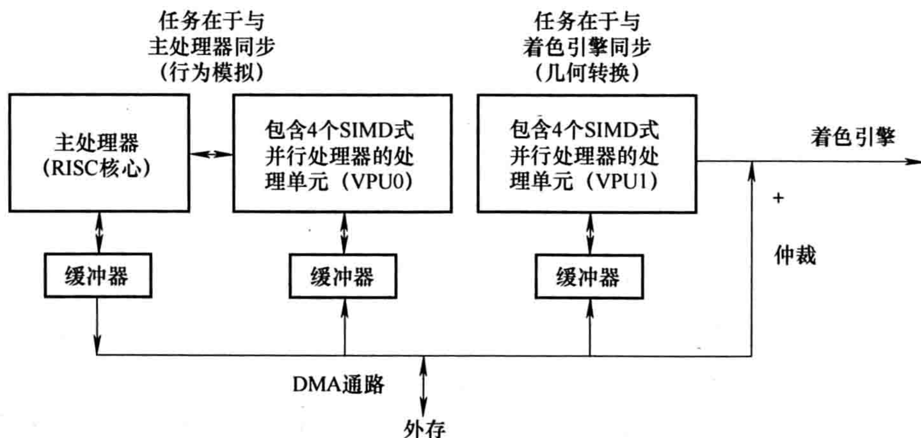


图 1.3 一个片上系统的高级功能视图：索尼 PlayStation 2[147,187]情感引擎

虽然本书的焦点是系统，但是为了理解系统，大家首先必须理解组件。所以，在回到讨论系统架构问题之前（本章稍后会讨论），先回顾一下组成系统的组件。

1.2　系统组件：处理器、存储器及互联

体系结构这个词表示系统的运作结构和用户对系统的视图。随着时间的推移，它已演变成同时包含功能描述和硬件实现两方面内容。系统架构定义了系统级的构建模块，如处理器、存储器及它们之间的互联。处理器架构决定了处理器指令集、相关编程模型及其详细实现。而详细实现可能包含不可见寄存器、分支预测电路和有关算术逻辑单元（Arithmetic Logic Unit，ALU）的具体细节。这种处理器的实现也被称为是微体系结构（见图 1.4）。

系统设计师需要同时拥有关于系统组件的程序员视角或用户视角，以及关于存储器、各种专用处理器和它们之间的互联的系统视角。接下来将涉及以下基本组件：处理器架构、存储器及总线或互联架构。

图 1.5 给出了一个基本的 SoC 模型。这些组件包括许多互联到一个或多个存储器上的异构处理器，以及一个可重构逻辑阵列。通常情况下，SoC 也包含用于管理传感数据和模-数转换或者用于无线数据传输的模拟电路。

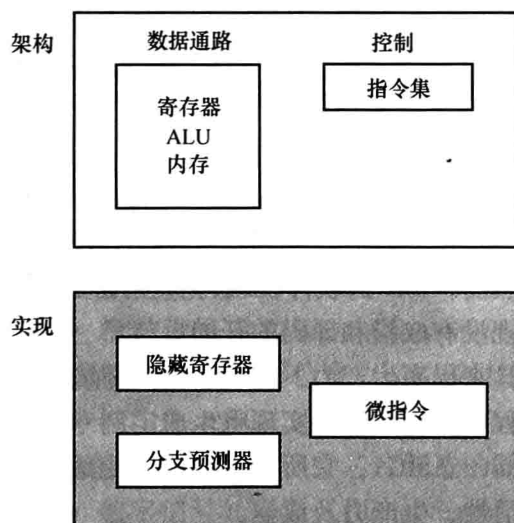


图 1.4　处理器架构及其实现

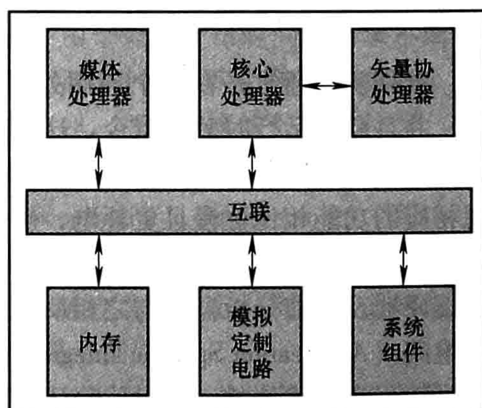


图 1.5　一个基本的 SoC 模型

例如，一个智能手机的 SoC 需要在传统电话的音频输入/输出功能之上提供互联网接入功能、用于视频通信的多媒体设备、文档处理，以及像游戏、电影这样的娱乐功能等。对于图 1.5 所示的一种可能配置包括用几个用于应用处理的 ARM Cortex-A9、一个用于媒体处理的Mali-400MP图像处理器实及一个 Mali-VE 视频引擎。这些系统组件和定制电路将与如摄像头、屏幕及无线通信部件这样的外设相互作用。这些元件可以用 AXI 总线（高级可扩展接口）接在一起。

如果所有的元件不能被集成在单个芯片上，最好的实现方式可能是将它们集成在一个主板系统上，但是通常来说，这也被称为 SoC。设计目标的专属特性是将板上系统（或 SoC）与附带板上存储的通用计算机相区别。设计之初，假设应用程序是被设计者熟知且特定的，在设计的过程中，就可以对元件进行选择、确定大小及评估。这样，针对一个目标应用，通过对选择、参数化和配置系统组件，就能够区分一个系统架构师和一个计算机架构师。

本章，主要看处理器的高层次定义——程序员视角或是指令集架构（Instruction Set Architecture，ISA），以及处理器微体系结构、存储层次和互联架构的基本内容。在后续章节，将学习关于这些元件实现问题的更多细节。

1.3 硬件和软件：可编程性与性能

在 SoC 设计中一个最基本的抉择就是选择各系统组件是用硬件还是用软件实现。表 1.1 给出了硬件实现与软件实现的优缺点。

表 1.1 硬件实现与软件实现的优缺点

	优　点	缺　点
硬件	速度快、功耗低	不灵活、适应性不强、不便于构建和测试
软件	灵活、适应性强、便于构建和测试	速度慢、功耗高

软件实现通常是在一个通用处理器（General Purpose Processor，GPP）上执行，并且是在运行时解释指令。这种架构有着良好的灵活性和适应性，而且在不同应用之间提供了一种共享资源的方式。然而，ISA 的软件实现与直接用硬件实现相应的功能相比更慢且更耗电，硬件实现没有取指和译码的开销。

大多数软件开发者使用高级语言和工具来提高生产力，如程序开发环境、优化编译器及性能分析器。与之相反，应用程序的直接硬件实现将生成定制专用集成电路（Application Specific Integrated Circuit，ASIC），它所提供的高性能是以损失可编程性为代价的，同时代价还包括灵活性、生产力及成本。

考虑到硬件和软件具有互补的特性，很多 SoC 设计都致力于将这两者各自的优点联合起来。一种显而易见的方法就是把性能关键的部分用硬件实现，其余部分则用软件实现。例如，如果一个占硬件执行时间 90% 的应用程序只占源代码 10% 的那部分，那么把这 10% 的部分用硬件实现的话能获得 10 倍的加速。这个方法将用到本书第 6 章的定制设计中。

GPP 上定制 ASIC 的硬件实现与软件实现可以看成是技术频谱上对可编程性与性能进行权衡的两个极端。有很多技术都处在这两个极端之间（见图 1.6）。这当中最负盛名的两个当属专用指令集处理器（Application Specific Instruction

Processor，ASIP）与现场可编程门阵列（Field Programmable Gate Array，FPGA）。

ASIP 是包含面向特定应用或领域的定制指令集的处理器。那些能在硬件上高效实现的定制指令通常会集成到一个包含基本指令集的基处理器上。这种实现方法通常可以用于改善标准指令集的传统实现方法，它能在完成同样任务的同时保留灵活性。本书第 6 章和第 7 章进一步探讨了一些涉及定制指令的问题。

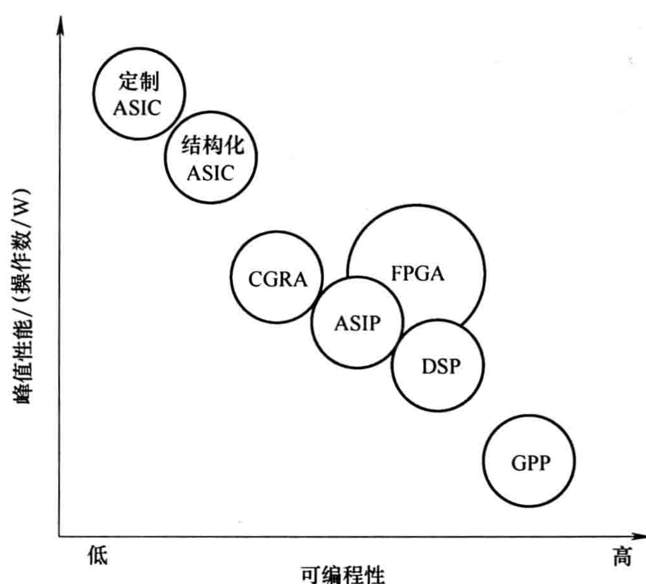


图 1.6　一个简化的技术比较：可编程性与性能

FPGA 通常包含运算部件阵列、存储器及它们的互联，并且这三者对应用构建者来说都是可编程的。FPGA 技术通常能提供一个很好的折中：更加灵活的同时速度比软件快，而且比定制 ASIC 硬件实现所需开发时间更短。就像 GPP 一样，它们作为现成设备用于编程而不需要经过芯片制造。由于缩短上市时间的需求不断上升及芯片制造的成本不断增加，FPGA 在实现数字设计领域正变得越来越流行。

大多数商用 FPGA 包含一个细粒度逻辑块阵列，每个逻辑块只有几位宽。它也可能包含如下部分：

• 粗粒度可重构架构（Coarse Grained Reconfigurable Architecture，CGRA）。它包含能处理位宽或多位宽数据的逻辑块，这种逻辑块可以形成数据通路的基石。

• 结构化 ASIC。它允许应用构建者在芯片制造之前自定义资源。虽然它提供的性能接近 ASIC，但是芯片制造的需求仍然是个问题。

• 数字信号处理器（Digital Signal Processor，DSP）。这些设备的组织和指令集针对数字信号处理应用进行了优化。就像微处理器一样，它们有固定的、不可重构的硬件架构。

这些技术的可编程性与性能各有优势（见图 1.6）。第 6 ~ 8 章提供了关于这些技术的深层次信息。

1.4　处理器架构

通常情况下，一款处理器或是以它们的应用或是以它们的架构（或结构）

作为特色，如表 1.2 和表 1.3 所示。对于一个应用，其需求空间总是很大，并且也有一系列的实现选项。因此，将一个特定的架构与一个特定的应用联系起来通常是很困难的。此外，有些架构联合了不同的实现方法，正如从 1.1 节 PlayStation 的例子所看到的那样。图像处理器包含一个四元件矢量处理功能部件（Function Unit，FU）SIMD 阵列。而其他 SoC 的实现包含使用超长指令字（Very Long Instruction World，VLIW）的多处理器，或还包含超标量处理器。

表 1.2　按照功能划分的处理器举例

处理器类型	应　用
图像处理单元（GPU）	三维图形；着色、渐变、纹理
DSP	通用，有时使用无线
媒体处理器	视频与音频信号处理
网络处理器	路由、缓存

表 1.3　按照架构划分的处理器举例

处理器类型	架构/实现方法
SIMD	单条指令应用到多功能部件（处理器）
矢量处理器（Vector Processor，VP）	单条指令应用到多流水寄存器
VLIW	编译器控制下每时钟周期发射多条指令
超标量	硬件控制下每时钟周期发射多条指令

从程序员的观点看，持续工作的处理器一次执行一条指令。然而，很多处理器通过流水线、多执行部件及多核技术，以对程序员透明的方式，具备了同时执行多条指令的能力。流水线是一种强大的技术，且被用于几乎所有的现代处理器实现中。而在编译时刻或是运行时刻提取并利用代码固有并行性的技术也被广泛使用。

利用程序并行性是计算机体系结构中最重要的目标之一。

指令集并行（Instruction Level Parallelism，ILP）是指在一个程序中多个操作能并行进行。ILP 能通过硬件、编译器或是操作系统技术来获得。在循环这一层，只要在相继的循环迭代之间没有数据相关，连续循环迭代是并行执行的理想候选。接着，在过程这一层，并行性的获得在很大程度上取决于程序所使用的算法。最后，多个独立的程序也能够并行地执行。

不同的计算机体系结构在构建时都利用了其固有的并行性。一般来说，一个计算机体系结构都是由一个或多个能同时操作的互联处理器元件（Processor Element，PE）组成的，用来解决单个整体性问题。

1.4.1　处理器：功能的观点

表 1.4 给出了不同的 SoC 处理器模型实例，包括不同的 SoC 设计及每种设计中所使用的处理器。对于这些实例，可以把它们定性为通用处理器或是支持游戏或信号处理应用的专用处理器。这种功能化的观点没有讲述关于底层的硬件实现。事实上，几种完全不同的架构方法都能实现相同的通用功能。以图像功能为例，需要具备着色、渲染和纹理功能及可能的视频功能。根据这些功能的相对重要性和所创建图像的分辨率，能够制造出完全不同的架构。

表 1.4　不同的 SoC 处理器模型实例

SoC	应　用	基本 ISA	处理器描述
Freescale e600[101]	DSP	PowerPC	超标量及矢量扩展
ClearSpeed CSX600[59]	通用	Proprietary ISA	包含 96 个处理部件的阵列处理器
PlayStation 2[147,187,237]	通用	MIPS	包含两个矢量协处理器的流水线处理器
ARM VFP11[23]	通用	ARM	可配置矢量协处理器

1.4.2　处理器：架构的观点

体系架构的角度至少以一种宽泛的方式描述了处理器系统的实际实现。而对于复杂的架构方法，需要更多细节才能理解其完整的实现。

简单顺序处理器　顺序处理器直接实现了顺序执行模型。这些处理器顺序地从指令流中处理指令。下一条指令只有在当前指令的所有执行都完成且结果被提交以后才能被处理。

指令的语义决定了动作序列必须产生特定结果（见图 1.7）。这些动作能够重叠，但是结果必须以特定的串行顺序出现。这些动作包括以下六项：

1）将指令取到指令寄存器（IF）；

2）对指令的操作码进行译码（ID）；

3）生成数据项的内存地址（AG）；

4）将操作数取到可执行寄存器（DF）；

5）执行特定的操作（EX）；

6）将结果写回到寄存器（WB）。

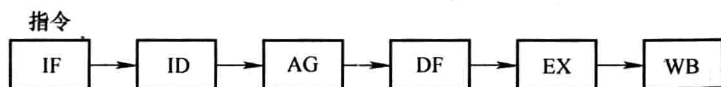


图 1.7　简单顺序处理器的指令时序

图1.8给出了一个简单顺序处理器模型。在执行期间，每个时钟周期里顺序处理器从指令流中执行一条或多条操作。指令就是一个容器，它代表了由处理器显式管理的最小执行包。一条指令包含了一个或多个操作。指令与操作之间的区别对于区分处理器行为至关重要。标量和超标量处理器每个周期消耗一条或多条指令，而每条指令只包含一个单一的操作。

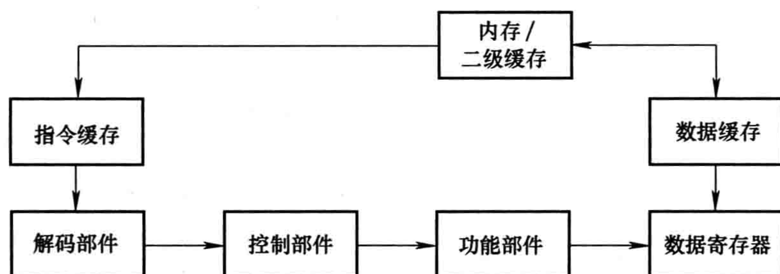


图1.8 一个简单顺序处理器模型

尽管概念上很简单，顺序地执行每条指令却有明显的性能缺陷：大量的时间都花在了重叠执行上而不是真正的执行中。因此，直接实现顺序执行模型获得了简易性的同时产生了明显的性能开销。

流水线处理器 流水线是一种利用并行性的简单方法，这种并行性是基于同时执行指令处理的不同阶段（取值、译码、执行等）而来的。流水线技术假设这些阶段在不同的操作之间是独立且可重叠的。即，当这种状况无法保持时，处理器通过暂停后续阶段来强制保持独立性。因此，多个操作能能够同时处理，每个操作处在指令处理的不同阶段。图1.9给出了流水线处理器的指令时序，假设这些指令是相互独立的。

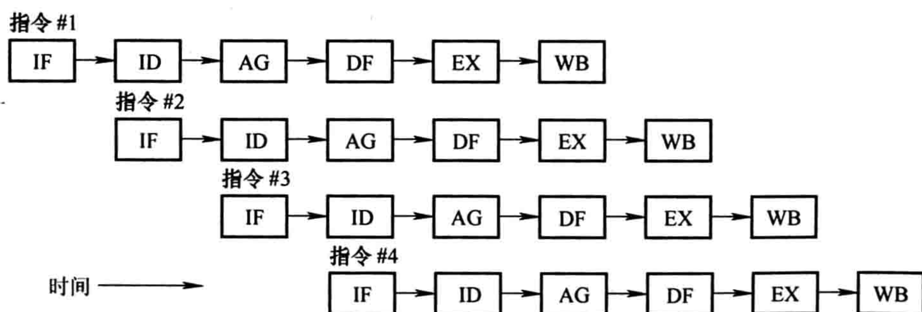


图1.9 流水线处理器的指令时序

对于一个简单流水线机器，在任意给定时刻，每个阶段只有一个操作。因此，一个阶段正在执行取指（IF），一个阶段正在执行译码（ID），一个阶段正在生成地址（AG），一个阶段正在访问操作数（DF），一个阶段正在执行

（EX），以及一个阶段正在回写结果（WB）。图 1.10 给出了一个简单流水线处理器模型。最古板的流水线形式一般被称为静态流水线，它要求处理器经过流水线的所有阶段而不管特定的指令是否需要这些阶段。而动态流水线允许根据指令的要求绕过一个或多个阶段。更复杂的动态流水线允许指令乱序完成，或乱序开始。乱序处理器必须确保程序的顺序一致性被保留下来。表 1.5 给出了使用流水线"软"处理器的 SoC 实例。

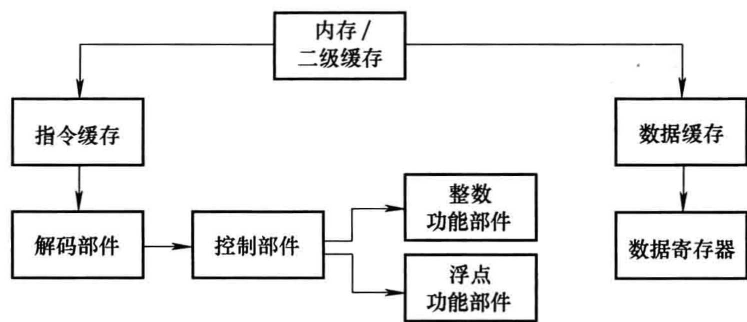


图 1.10　一个简单流水线处理器模型

表 1.5　使用流水线"软"处理器的 SoC 实例[177,187]

（软处理器是由 FPGA 或是类似的可重构技术实现的）

处 理 器	字长/bit	流水线段数	指令缓存/数据缓存①总量/KB	FPU②	常 用 目 标
Xilinx MicroBlaze	32	3	0~64	可选	FPGA
Altera Nios Ⅱ fast	32	6	0~64	—	FPGA
ARC 600[19]	16/32	5	0~32	可选	ASIC
Tensilica Xtensa LX	16/24	5~7	0~32	可选	ASIC
Cambridge XAP3a	16/32	2	—	—	ASIC

① 指可配置的指令缓存与数据缓存。
② FPU：Floating Point Unit，浮点部件。

ILP　流水线虽然并不必然导致在同样的时间执行多条指令，但是有其他的技术能够做到。这些技术可以联合静态调度与动态分析，来同时执行几个不同操作的实际求值阶段，这样可能产生大于每周期一个操作的执行速率。由于历史上大多数指令都只包含一个单一的操作，这种类型的并行被称为 ILP。

超标量处理器和 VLIW 处理器就是利用 ILP 的两种架构。它们使用不同的技术来获得大于每周期一个操作的执行速率。超标量处理器动态地检查指令流来确定哪些操作是独立且可执行的。而 VLIW 处理器依赖编译器来分析可用操作（OP），并将独立操作安排成宽指令字。这样就可以并行执行这些操作而无需进一步分析。

图 1.11 给出了流水线化的 ILP 处理器的指令时序，它是流水线化的超标量处理器或是 VLIW 处理器每周期执行两条指令的指令时序。在这种情况下，所有的指令都是独立的，这样它们能够并行执行。下面将会描述这两种架构的更多细节。

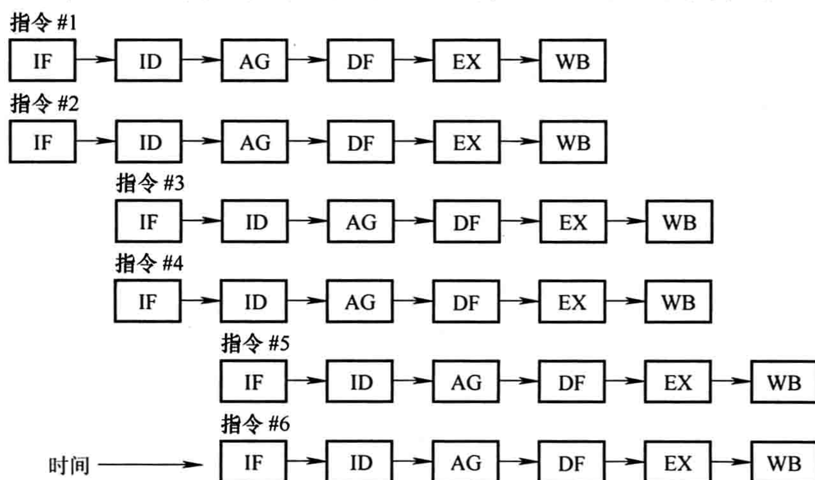


图 1.11 流水线化的 ILP 处理器的指令时序

（1）超标量处理器 动态流水线通过其标量特性的优点仍然仅限于每周期执行单条操作。这种限制能够通过增加多个功能部件及动态调度器来避免，从而使得每周期能处理多条指令（见图 1.12）。这些超标量处理器[135]能够获得每周期多条指令的执行速率（通常限制为 2，但是根据应用类型还可能更多）。超标量处理器最显著的优势在于每周期处理多条指令对用户来说是透明的，并且还在于它能在获得更好性能的同时具备二进制代码兼容性。

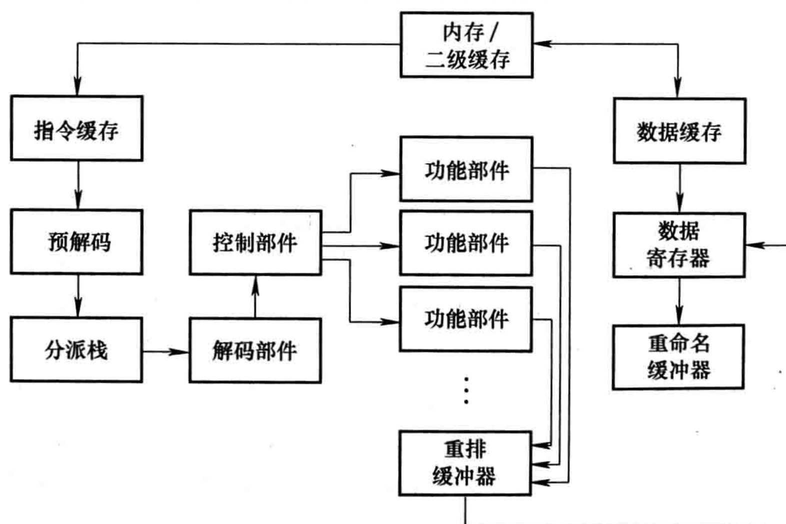


图 1.12 超标量处理器模型

对比于动态流水线处理器，超标量处理器增加了一个调度指令窗口，它能在每个周期从指令流中分析多条指令。尽管这些指令是并行处理的，但是对待它们与在流水线处理器中是一样的。在指令发射执行之前，硬件必须检查该指令与前面指令的依赖关系。

由于动态调度逻辑的复杂性，高性能超标量处理器都限制在每周期只处理 4 ~6条指令。尽管超标量处理器能够从动态的指令流中利用 ILP，但是要想利用更高级别的并行性需要其他的方法。

（2）VLIW 处理器　相比于通过硬件的动态分析来决定哪些指令能并行执行，VLIW 处理器（见图 1. 13）需要依靠编译器的动态分析。

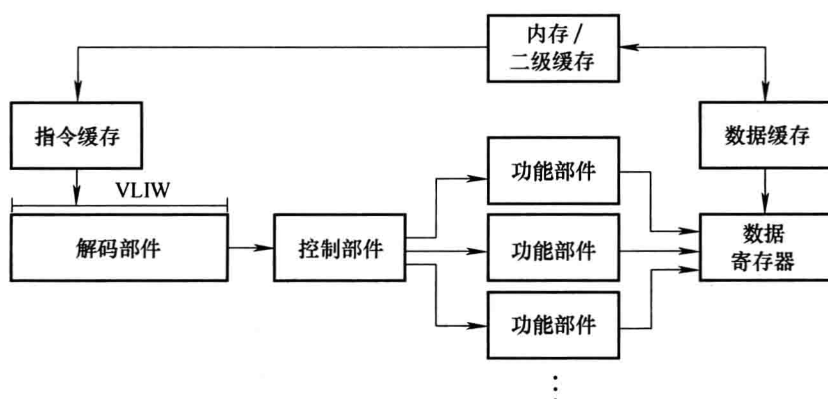


图 1. 13　VLIW 处理器模型

因此，VLIW 处理器没有超标量处理器复杂，并且拥有提升性能的潜力。VLIW 处理器从经过静态调度且包含多个独立操作的指令中执行操作。由于 VLIW 处理器的控制复杂性没有显著高于标量处理器，因而其提高的性能不是因为复杂性的增加而换来的结果。

VLIW 处理器依赖编译器的静态分析，而且没法利用任何动态执行的特点。对于那些能够通过静态调度来高效地使用处理器资源的应用来说，一个简单的 VLIW 实现就能产生很高的性能。然而不幸的是，不是所有的应用都能高效地静态调度。很多应用不是完全按照编译器中的代码调度器所安排的路径来执行的。如下两类执行变化会影响已调度的执行行为：

1）延迟了的操作结果，其延迟不同于编译器调度后所设想的延迟。

2）例外或中断，使得执行路径变成完全不同的和预料之外的代码调度。

尽管暂停处理器能够控制延迟了的结果，但是这样会明显导致性能损失。最常见的执行延迟当属数据缓存失效。很多 VLIW 处理器通过避免数据缓存和设想操作的最坏延迟来避免所有会产生延迟的情形。然而，当没有足够的并行性来隐藏暴露出来的最坏操作延迟时，指令调度会产生很多未完全填充满的指令甚至是

空指令，进而导致很差的性能。

表1.6和表1.7给出了一些代表性的超标量处理器和VLIW处理器SoC实例。

表1.6 使用超标量处理器的SoC实例

设　　备	功能部件数目	发射宽度	基本指令集
MIPS 74K Core[183]	4	2	MIPS32
Infineon TriCore2[129]	4	3	RISC
Freescale e600[101]	6	3	PowerPC

表1.7 使用VLIW处理器的SoC实例

设　　备	功能部件数目	发射宽度
Fujitsu MB93555A[103]	8	8
TI TMS320C6713B[243]	8	8
CEVA-X1620[54]	30	8
Philips Nexperia PNX1700[199]	30	5

SIMD架构：阵列与矢量处理器　SIMD类的处理器架构既有阵列处理器又有矢量处理器。SIMD处理器能够自然地响应使用如矢量和矩阵这样特定规则的数据结构。从汇编级别程序员的角度看，除了有些操作是针对集合数据进行运算以外，对SIMD架构编程看起来很类似对简单处理器编程。由于这样的规则结构广泛使用于科学编程中，SIMD处理器在这些环境中变得越来越成功。

最流行的两种SIMD处理器当属阵列处理器与矢量处理器。在实现与数据组织方面，两者都不相同。阵列处理器是由很多互联的处理器元件组成的，每个处理器元件都拥有其自己的本地存储空间。而矢量处理器是由单个处理器组成的，它能引用一个全局的存储空间并且拥有能进行矢量操作的特殊功能部件。

阵列处理器或是矢量处理器能够通过扩展其他常规机器的指令集来得到。这些扩展的指令能够控制处理器或是某种协处理器中的特殊资源。这样扩展的目的是为了提高特殊应用的性能。

（1）阵列处理器　阵列处理器（见图1.14）是由一组并行处理器元件通过一个或多个网络连接起来的，而且很可能包含本地和全局的元件间通信和控制通信。处理器元件通过锁步操作来响应控制处理器（如SIMD）的单广播指令。每个处理器元件（Processor Element，PE）有其私有的内存，而数据则以一种规则的方式分配并穿过元件，这种方式既依赖数据的实际结构，还依赖在数据上所执行的运算。直接访问全局内存或是另一个处理器元件的内存开销是很大的，所以中间值是通过阵列及本地处理器间连接来传播的。这就需要数据能小心地分配，使得传播值的路由能够简单而规则。比起支持处理器元件间复杂或是不规则的数

据路由，有时复制值然后运算会更容易。

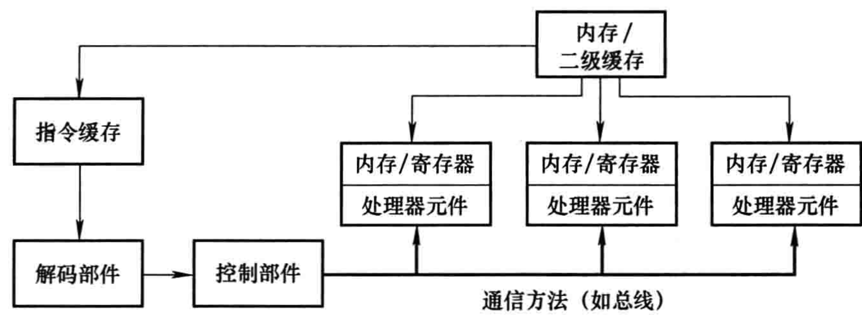


图 1.14 阵列处理器模型

由于指令是广播的，本地处理器元件没办法改变指令流的流动。然而，个别处理器元件能根据本地状态信息有条件地禁用指令。即，当这种状况发生时，这样的处理器元件是空闲的。实际的指令流是由一个以上的固定操作流组成的。阵列处理器通常与一个通用控制处理器耦合在一起。通用处理器既提供标量操作也提供数组操作。而数组操作会被广播到阵列中的所有处理器元件。这个控制处理器执行应用的标量部分、与外界交互，并控制执行流。而阵列处理器则执行由控制处理器所指定的应用的数组部分。

一个能在阵列处理器上使用的合适应用需具有以下几个关键特征：大量的具有规则结构的数据、均匀地施加到许多或是所有数据集上的运算、与运算和数据相关的简单而规则的模式。求解 Navier-Stokes 方程就是一个具备这些特征的应用实例，然而任何包含显著矩阵运算的应用都很可能从阵列处理器的并发能力中获益。

表 1.8 给出了基于阵列处理器的 SoC 实例。ClearSpeed 处理器就是一个面向信号处理应用的阵列处理器芯片的实例。

表 1.8 基于阵列处理器的 SoC 实例

设　　备	每个控制部件的处理器数目	数据大小/bit
ClearSpeed CSX600[59]	96	32
Atsana J2211[174]	可配置	16/32
Xelerator X10q[257]	200	4

（2）矢量处理器　矢量处理器就是个单一的类似传统单一流处理的处理器，只不过它的某些功能部件（和寄存器）是对矢量进行操作的。所谓矢量操作是指数据值序列看起来是当做一个整体进行操作的。这些功能部件都是深流水且时钟频率很高。相比于标量功能部件，尽管矢量流水线的延迟更高，但是快速输入矢量数据交付加上高时钟频率，使得其吞吐量非常大。

现代矢量处理器要求矢量明确地加载到特殊矢量寄存器且要存回内存——基于相似的理由与现代标量处理器的做法一样。矢量处理器有几个特点使得它们拥有很高的性能。一大特点就是当其在矢量寄存器中运算时，拥有能在矢量寄存器与主存之间并发加载和存储值的能力。这是一个很重要的特点，由于矢量寄存器的长度限制，这就需要那些比寄存器长度还长的矢量能够分段处理。所谓分段是一种称为露天采矿的技术（strip mining）。而内存访问和运算如果不能同时发生就会造成明显的性能瓶颈。

大多数矢量处理器都支持结果旁路，在这种情况下称为链接。即一旦前面的运算的第一个结果可用以后，允许后续的运算马上开始。因此，不需要等待全部矢量都处理完成，后续运算能够与前面相互独立的运算同时执行。连续的运算能够高效地复合操作就像它们是单条操作一样，这样总的延迟就是第一条操作的延迟加上剩下操作的流水线延迟和链接延迟，而不是无链接情况下从头到尾的开销。例如，除法能通过将倒数与乘操作链接来进行综合。链接通常适用于加载操作及常规运算的结果。

典型的矢量处理器配置（见图 1. 15）由一个矢量寄存器堆、一个矢量加法部件、一个矢量乘法部件及一个矢量倒数部件（用于和矢量乘法部件结合来进行除法运算）组成。矢量寄存器堆包含多个矢量寄存器（元件）。

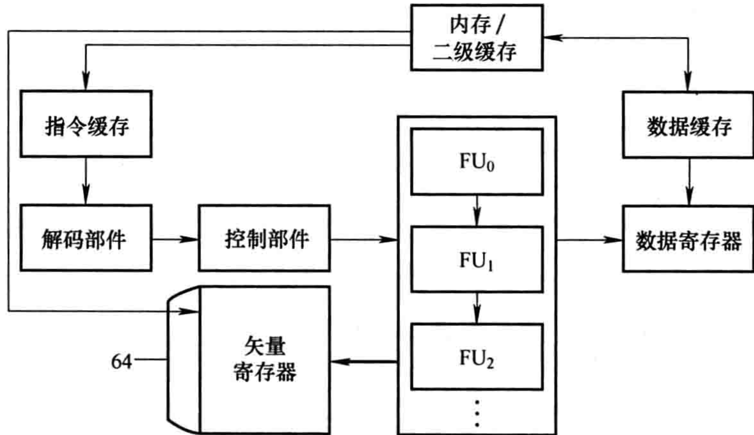


图 1. 15 矢量处理器模型

表 1. 9 给出了使用矢量寄存器的实例。IBM 大型机把矢量指令（及硬件支持）作为选项提供给特定用户。

表 1. 9 使用矢量处理器的 SoC 实例

设 备	矢量功能部件	矢量寄存器
Freescale e600[101]	4	32 个可配置
Motorola RSVP[58]	4（64 位可分成 16 位）	两个流（两入一出）存储器
ARM VFP11[23]	3（64 位可分成 32 位）	4 ×8，32 位

多处理器　通过某种形式的互联来共享结果，多个处理器能够合作运行来解决一个单一的问题。在这种配置中，尽管每个处理器完全独立地运行，但是大多数应用在执行期间需要某种形式的同步，使得处理期间能够传递信息和数据。由于多个处理器共享存储且执行单独的程序任务（MIMD⊖），这样它们的实现会比阵列处理器复杂得多。大多数配置对所有处理器来说都是相同的，尽管这并不是一个必须要求。表 1.10 给出了多处理器与多线程处理器的 SoC 实例。

表 1.10　多处理器与多线程处理器的 SoC 实例

SoC	Machanick[162]	IBM Cell[141]	Philips PNX8500[79]	Lehtoranta[155]
CPU 数目	4	1	2	4
线程	1	很多	1	1
矢量部件	0	8	0	0
应用	各种各样	各种各样	HDTV	MPEG 解码
注释	仅为建议方案		又称为 Viper 2	软处理器

多处理器中的互联网络在处理器元件间传递着数据并同步独立执行流。当处理器内存被分配给所有处理器且只有本地处理器元件能访问它时，所有的数据共享都要明确地使用消息来进行，而且所有的同步也是在消息系统中处理的。当处理器内存被所有处理器元件共享时，同步是个大问题。当然，可以通过存储系统在处理器元件之间用消息来传递数据和信息，但是这样做必然没有最高效地使用系统。

当处理器元件之间的通信是通过一个共享的存储地址空间进行时，处理器元件之间不管是全局式共享还是分布式共享（称为分布式共享存储是为了区别于分布式存储），都会产生两个显著的问题。第一个问题就是维持存储一致性：在内存引用上对程序员可见的影响，既来自于处理器元件内部还来自不同处理器元件之间。这个问题通常是通过硬件和软件技术的联合来解决的。第二个问题就是缓存相关性，这是个程序员不可见机制，确保所有处理器元件对于给定的存储部件所看到的都是相同的值。这个问题通常是专门通过硬件技术来解决的。

多处理器系统最主要的特点就是存储地址空间的特性。如果每个处理器元件都有其自己的地址空间（分布式存储），那么处理器元件之间唯一的通信方法就是通过消息传递。而如果地址空间是共享的（共享式存储），那么通信就可以通过存储系统进行。

⊖ MIMD：Multiple Instruction Stream Multiple Data Stream，多指令流多数据流。

当把存储一致性和缓存相关性考虑在内时，分布式存储机器的实现远比共享式存储机器的实现要容易。然而，对分布式存储处理器编程会更加困难，因为应用必须能够利用消息传递这种处理器元件间唯一的通信形式且不能被其限制。另一方面，尽管有维持一致性和相关性方面的问题，但是对共享式存储处理器进行编程时能够利用任何与给定通信要求相适应的通信模式，因而也更容易编程。

1.5 内存与寻址

不同的SoC应用对内存的需求各不相同。有的情况下，内存结构能简单到程序完全驻留在只读内存（Read Only Memery，ROM）中，而数据则在片上RAM中。而另一种情况下，内存系统可以通过内存管理及缓存层次来支持复杂的需要片外存储空间（SoC）的操作系统。

为什么不是简单地把内存与处理器包含在芯片上呢？这其中有很多妙处：

1）提高了内存的可达性，既改善了内存的访问时间，又提高了内存的访问带宽。

2）减少了对大缓存块的需求。

3）提高了内存密集型应用的性能。

但是这样做也存在问题。第一个问题就是DRAM内存处理技术不同于标准的微处理器处理技术，而且会导致在位密度上有所牺牲。第二个问题更加严重，如果内存被处理器芯片所限制，其大小也会相应地被限制。这样一来，那些对实际内存空间需求量非常大的应用就会“跛脚”了。因此，常规的处理器芯片模型就演变成（见图1.16）实现多个强大的同类处理器在其自己的多片模块上与片外主存共享高达两级或三级缓存。

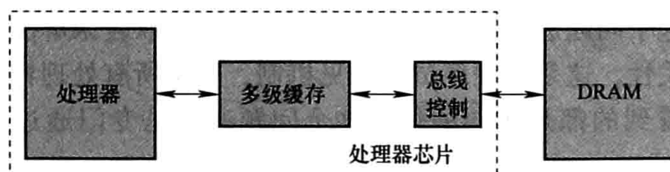


图1.16 处理器加片外内存

从设计复杂性的观点看，它有着成为“万能”解决方案的优势：一种实现就适合了所有的应用，尽管它没有肯定达到一样好。所以，当大量的设计工作都需要这种实现时，产量能够大到足以证明成本是可合理摊销的。

对于这种方法的替代品也是很明显的。对于那些内存大小有界的特定应用，

能够实现一种内存集成式的 SoC，如图 1.17 所示（还可以回想图 1.3 所示的结构）。

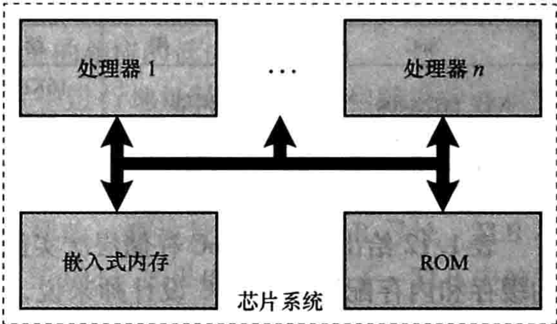


图 1.17　SoC：处理器与内存

一个相关但又独立的问题是，应用需要虚拟存储（把磁盘空间映射到内存）吗？或是全部都用实际内存是合适的吗？下面将介绍对虚拟存储寻址的需求。

最后，内存可以是集中式的也可以是分布式的。即使在这里，尽管内存是在几个分布式的模块中实现的，对程序员来说内存仍然可以显得是单个（集中式）共享内存。表 1.11 给出了几种 SoC 内存考虑事项。

表 1.11　几种 SoC 内存考虑事项

问　题	实　现	注　释
内存位置	片上 片外	大小有限且固定 板上系统，访问慢，带宽有限
寻址	实际寻址 虚拟寻址	大小有限，OS 简单 非常复杂，需要 TLB 及指令有序执行的支持
分类（按多处理编程方式）	共享存储 消息传递	需要硬件支持 额外编程
分类（按实现方式）	集中式 分布式	受限于芯片考量 可与处理器或其他存储模块聚合

存储系统是由各个存储层次中的物理存储元件组成的。这些元件包括那些由指令集所指定的（寄存器、主存及磁盘扇区），以及那些对用户程序大都透明的（缓冲寄存器、缓存及页式虚存）。

1.5.1　SoC 内存实例

表 1.12　不同 SoC 存储层次实例

SoC	应　用	缓存大小	片上/片外	实际/虚拟
NetSilicon NET + 40[184]	网络	4KB 指令缓存，4KB 数据缓存	片外	实际
NetSilicon NS9775[185]	打印	8KB 指令缓存，4KB 数据缓存	片外	虚拟

（续）

SoC	应　　用	缓存大小	片上/片外	实际/虚拟
NXP LH7A404[186]	网络	16KB 指令缓存，4KB 数据缓存	片上	虚拟
Motorola RSVP[58]	多媒体	瓦片缓冲存储器	片外	实际

表 1.12 给出了不同 SoC 存储层次实例，展示了很多不同的 SoC 设计及它们的缓存和内存配置。对 SoC 设计师来说，把 RAM 和 ROM 放在片上还是片外是一个值得考虑的重要问题。表 1.13 给出了嵌入了存储器宏单元的 SoC 实例，展示了各种嵌入了存储器宏单元的 SoC 例子。

表 1.13　嵌入了存储器宏单元的 SoC 实例（单元类型的讨论详见本书第 4 章）

厂　　商	单元类型（典型）	SoC 用户（典型）
Virage Logic	6T（SRAM）	SigmaTel/ARM
ATMOS	1T（eDRAM）	Philips
IBM	1T（eDRAM）	IBM

注：T 代表一位单元上的晶体管数目。

1.5.2　寻址：内存架构

用户的内存视图主要是指对程序员来说可用的寻址设备。这些设备有些是对应用程序员可用的，而有些是对操作系统程序员可用的。虚拟存储使得那些内存需求比物理内存还要大的程序得以运行，并且多个应用程序执行时允许有分开的寻址空间以便防止未授权内存访问。当虚拟寻址设备被正确地实现和编程使用后，内存就能高效而安全地访问了。

虚拟存储通常是由一个存储管理部件所支持的。概念上讲，物理内存寻址是由以下连续的（至少）三步所决定的：

1. 应用中产生一个进程地址。伴随着进程 ID 或用户 ID，它定义了虚拟地址。虚拟地址 = 偏移 +（程序）基地址 + 索引，偏移是在指令中指定的，而基地址与索引的值都在特定的寄存器中。

2. 由于多个处理器必须在同一个内存空间里合作，所以进程地址需要调整并浮动。这通常根据段表来完成的。虚拟地址的高位用于段表寻址，段表中有进程（预订的）基地址和界限值，这样形成了系统地址。系统地址 = 虚拟地址 +（进程）基地址。系统地址必须小于界线。

3. 虚拟与实际。对很多 SoC 应用（以及所有的通用系统）来说，内存空间超过了（实际）实现的内存。在这里，存储空间是在磁盘上实现的，且只有那些最近使用的区域（页面）才会放到内存来。这些可用的页面是由页表定位的。

当页数据从磁盘加载以后，其在内存中的位置将会作为“实际”或物理内存地址的高位。而实际地址的低位与虚拟地址的相应的低位是一样的。

通常，表（段表或页表）是在内存中进行地址转换的，并且转换中必须使用一种称为旁路转换缓冲（Translation Lookaside Buffer，TLB）的机制来加快转换。TLB 就是一个简单的寄存器系统，通常是由 64 ~ 256 项组成的，保存着用于重用的最近地址转换。只有少量的（散列的）虚地址位用于 TLB 寻址。TLB 项中既有实际地址也有完整的虚地址（及其 ID）。如果虚地址匹配了，TLB 中的实际地址就可用了。否则，就出现了 TLB 失效事件，然后就必须进行一个完整的址转换了（见图 1.18）。

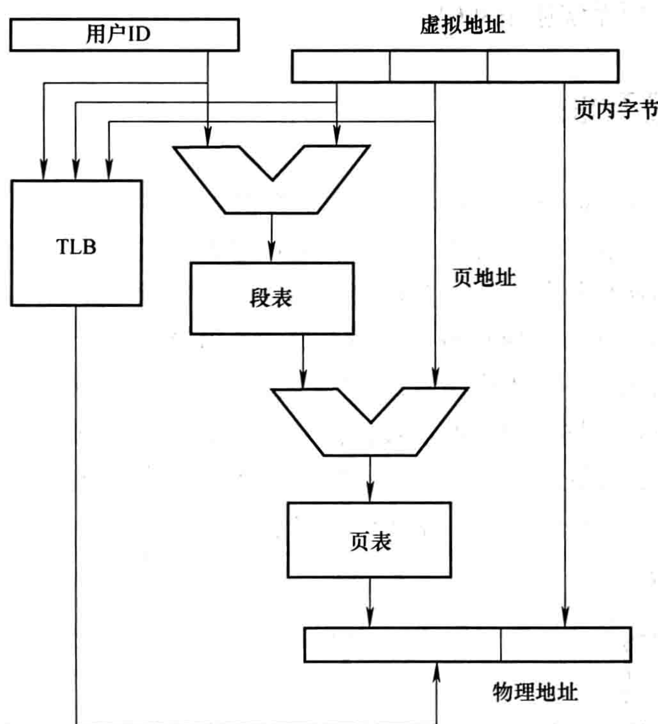


图 1.18　包含 TLB 旁路的虚实地址映射

1.5.3　SoC 操作系统内存

操作系统及其内存管理功能是涉及 SoC 设计的关键的抉择（或要求）之一。设计师的主要兴趣在于对虚存的需求。如果系统受限于实际内存（物理寻址的，而不是虚拟寻址的）且内存大小只有数十兆字节，那么系统可以实现成片上（完全片上内存）真实系统。而虚拟存储需要复杂的存储管理部件，通常较慢而且明显更贵。表 1.14 给出了 SoC 设计中的操作系统，展示了一些当前的 SoC 设计及其操作系统。

表 1.14 SoC 设计中的操作系统

OS	厂　商	存储器类型
uClinux	开源的	实际
VxWorks（RTOS）[254]	美国风河（Wind River）公司	实际
Windows CE	美国微软（Microsoft）公司	虚拟
Nucleus（RTOS）[175]	美国明导（Mentor Graphics）公司	实际
MQK（RTOS）[83]	美国 ARC 公司	实际

当然，快速的实际内存设计伴随功能上的代价。用户只能以有限的方式来创建新的进程及扩展系统应用基础。

1.6 系统级互联

SoC 技术通常依赖各种预先设计好的电路模块（即 IP⊖块）的互联来形成一个完整的并能集成在单个芯片上的系统。通过这种方式，设计任务从电路级上升到系统级。此时所使用的互联方法就会成为系统级性能与成品可靠性所关注的重点。设计良好的互联方案应具备有力而高效的通信协议，并明确地定义成发布标准。这样不同组织、不同人所设计的 IP 块之间就能方便地互通而且能够促进 IP 块重用。它应能在不同模块之间能提供高效的通信以获得最大的并行程度。

SoC 互联方法主要分为两类——总线与片上网络，如图 1.19 和图 1.20 所示。

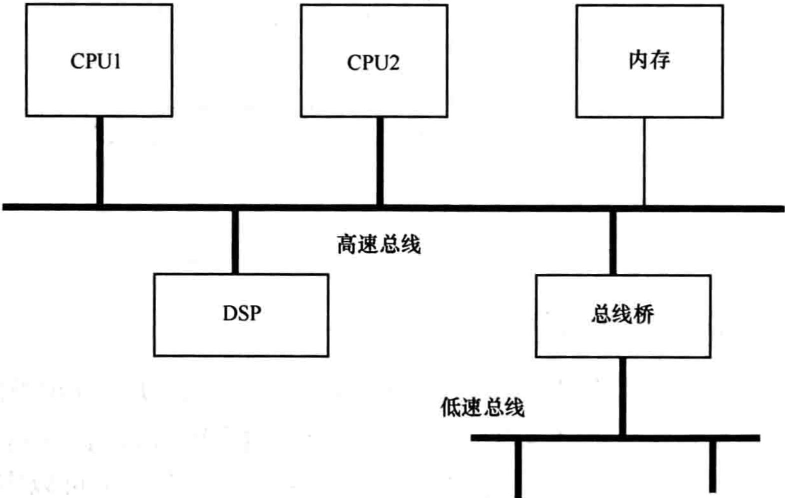


图 1.19 SoC 系统级互联：基于总线的方法

⊖ IP：Intellectual Property，知识产权。

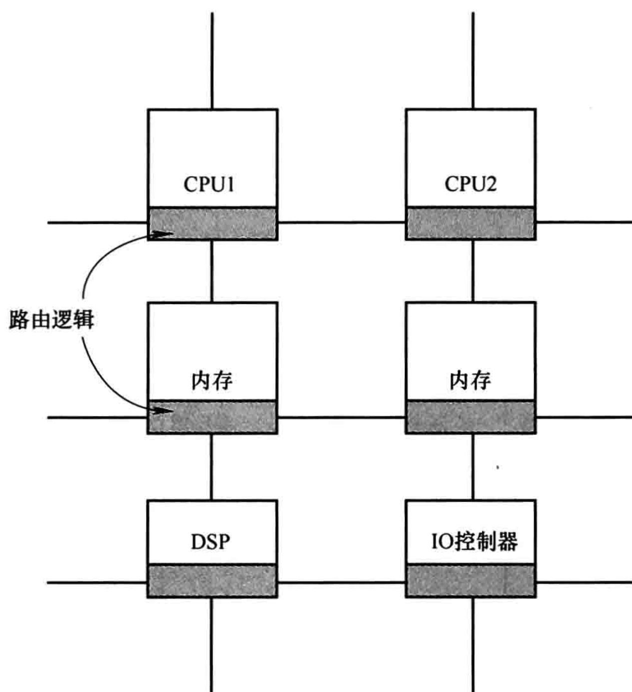


图 1.20　SoC 系统级互联：片上网络方法

1.6.1　基于总线方法

在基于总线的方法中，IP 块设计成符合已发布的总线标准（如英国 ARM 公司的 AMBA⊖[21]或是美国 IBM 公司的 CoreConnect[124]）。模块之间的通信是通过物理互联的地址、数据及控制总线信号的共享来获得的。这是 SoC 系统级互联中常见的一种方法。通常，系统会通过层次化样式来使用两个或更多的总线。为了优化系统级的性能与开销，最靠近 CPU 的总线具有最高的带宽，而离 CPU 最远的总线有着最低的带宽。

1.6.2　片上网络方法

片上网络系统是由开关阵列组成的，这些开关要么像交叉开关矩阵那样动态地开关，要么像网格网络那样静态地开关。

交叉开关矩阵方法使用异步通道来连接同步的能以不同时钟频率运作的模块。这种方法相比于基于总线的系统有着更高的吞吐量，同时也让系统与多时钟域的集成更加容易。

⊖ AMBA：Advanced Microcontroller Bus Architecture，高级微控制器总线架构。

在简单的静态开关网络（见图 1.20）中，每个结点都包含形成核心的处理逻辑及其自己的路由逻辑。互联方案是基于一个二维的网格拓扑。开关之间的所有通信是通过数据包进行的，而且是通过各自结点内部的路由接口电路来路由的。由于开关之间的互联有着固定的距离，就大大减少了如导线延迟及串扰噪声这样的互联相关问题。表 1. 15 给出了不同 SoC 的互联模型实例。

表 1. 15 不同 SoC 的互联模型实例

SoC	应　用	互联类型
ClearSpeed CSX600[59]	高性能计算	ClearConnect 总线
NetSilicon NET + 40[184]	网络	定制总线
NXP LX7A404[186]	网络	AMBA 总线
Intel PXA27x[132]	移动/无线	PXBus
Matsushita i-Platform[176]	媒体	内联总线
Emulex InSpeed SoC320[130]	开关	交叉开关
MultiNOC[172]	多处理系统	片上网络

1.7 SoC 设计方法

为了有效且高效地完成设计，弄清需求与规范、在设计的不同阶段进行迭代是设计过程中两个重要的思想。

1.7.1 需求与规范

需求与规范是任何系统设计情形中基本的概念。在设计开始前两者都必须彻底理解清楚。在设计过程开始和结束时它们很有用：在开始时，用于阐明需要获得什么；在结束时，作为参考来对完整的设计进行评价。

系统需求大部分是外部生成的系统准则。它们可能来自于竞争、销售观点、客户需求、产品盈利分析或是以上的综合。系统需求很少是简洁的或是明确的。事实上，需求经常还可能是不现实的："我要它快，我要它便宜，我立刻就要它！"

小心地分析需求表达，并花足够的时间了解市场情形对设计师来说很重要，这样才能判定出需求所表达的所有因素及这些因素所暗示的优先级。设计师在需求判定中考虑的一些因素：

- 与原先设计或是已发布标准的兼容性；

- 原先设计的重用；
- 客户需求、投诉；
- 销售报告；
- 成本分析；
- 竞争设备分析；
- 原先产品或是竞争产品的问题报告（可靠性）。

设计师也可以基于一些在类似系统环境还未使用过的新技术、新思想或是新材料来引进新需求。

系统规范是目标系统设计的量化与优先准则。设计师获得需求后必须形成最终系统的简洁而明确的说明集。设计师可能不知道最终系统是什么样子的，但是通常脑中会有个“稻草人”的设计，它似乎可提供符合规范的可行框架。在任何有效设计过程中，如果最终设计明显像稻草人设计的话着实会令人大吃一惊。

规范并没有完成设计过程的任何部分，它只是对过程进行了初始化。现在就可以开始组件和方法选择、替代研究及系统成分的优化了。

1.7.2 设计迭代

设计总是一个迭代的过程。所以，如何开始第一步最初的设计是一个很明显的问题。正是通过这最初的设计来按照设计准则进行迭代和优化。为了我们的目标，定义了几种基于设计阶段的设计类型。

初始设计 在其他性能和成本准则还没有考虑的时候，这是用来展示符合关键需求的第一个设计。例如，处理器、存储器或是输入/输出（I/O）应该在尺寸上符合优先级高的实时性约束。重要组件及其参数需要精心选择和分析，以便对所期望的理想性能和成本有个好的认识。理想化的并不意味着就是不切实际的，而是指所期望的使用和运算面积或是数据带宽容量的简化模型。它通常是个简单的线性性能模型，如处理器所期望的每秒百万条指令（Million Instructions Per Second，MIPS）速率。

优化设计 一旦基本性能（或面积）需求符合要求且基本功能确保了以后，目标就变成最小化成本（面积）及功耗或是完成设计所需的设计工作量。接着就是设计过程中的迭代步骤。设计过程第一步是使用高保真工具（模拟、实验布局等）来确保初始设计确实满足了设计需求与规范。后续步骤是根据设计准则精练、完成并改善设计。

图 1.21 给出了 SoC 初始设计过程。这个设计在细节上足以创建出设计的组件视图及对组件期望性能的相应推测。在这一步，这个推测必然是简化了的且参考于理想化的组件视图（见图 1.22）。

1. 了解功能、成本及实时要求。
2. 确认设计必须要符合的关键需求。
3. 基于对关键需求的影响，给处理器、内存及互联组件的选择确定优先级。
4. 评估关键需求是否已满足。
5. 如果是，那么初始设计已完成。
6. 如果不是，那么尝试选择不同的组件，再回到第3步。

图 1.21　SoC 初始设计过程

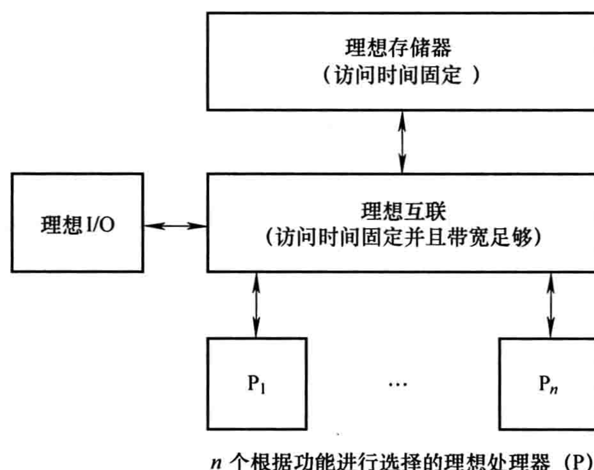


图 1.22　理想 SoC 组件

系统性能是由组件的最小容量所限制的。其他组件通常仅是在关键组件上呈现一个延迟来进行建模。在一个好的设计中，限制系统性能的组件才是最昂贵的组件。系统处理事务的能力应该紧跟限制组件的能力。通常，这个限制组件就是处理器或存储器复合设备。

通常，设计要么是由①一个使得功能和成本变得很重要的特定实时需求；要么是由②成本-性能约束下的功能或吞吐量来驱动的。在情况①中，实时约束是根据 I/O 考虑所提出的，处理器-存储器-互联系统必须符合要求。I/O 系统决定了性能，系统的剩余过量容量通常用于给系统增加功能。在情况②中，目标在于最小化成本的同时提高任务吞吐量。吞吐量是由受到最大约束的组件所限制的，所以设计师必须充分了解折中点。在这些设计中这样做更具灵活性，而且相应地在决定最终设计方面更有选择性。

本书的目标在于通过以下内容来提供用于决定最初设计的方法：

（a）描述组件范围——处理器、存储器及互联，可用于 SoC 构建；

（b）提供各种应用领域需求的实例，如数据压缩和加密；

（c）阐明初始设计或报告实现如何能符合特定需求。

之后将阐释这种方法，本书第 3 ~5 章通过组件成分来涉及（a）部分，而在本书第 6 章涉及系统配置与定制方面的技术。

正如原先所提到的那样，设计师必须优化处理和存储方面的各个组件。这种优化过程需要大量的模拟。而通过相关网站来提供基本的模拟工具。

1.8　系统架构及其复杂性

处理器架构与系统架构的基本不同点在于系统增加了另外一层复杂性，并且这些系统复杂性限制了成本节约。历史上，计算机的概念就是单个处理器加上内存。如果这种概念一直不变（还在板卡的限度内），将处理器实现在一个或多个硅片上都没有改变设计复杂性。而一旦芯片密度允许标量处理器装在一个芯片上，那么复杂性问题就改变了。

假设需要 100 000 个晶体管及一个小一级缓存来实现一个 32 位流水线处理器，这里称其设计复杂性大小为一个处理器单元。

如果需要实现 100 000 个晶体管组成的处理器，增加的片上晶体管密度不会很影响设计复杂性。每块芯片上有更多晶体管时，在增加芯片复杂性的同时，也简化了组成处理器的多个芯片之间的互联问题。一旦将单元处理器实现在单个芯片上，设计复杂性问题就改变了。而在这之后随着晶体管密度显著地增加，也会有很多显而易见的处理器延伸策略来改善性能：

1）增加缓存。在这里增加缓存存储及二级缓存，因为随着缓存变大，访问时间就变慢了。

2）更高级处理器。可以实现超标量处理器或是 VLIW 处理器，这样每个周期就可以执行多条指令。此外，可以加速那些影响关键路径延迟的部件，尤其是浮点执行时间。

3）多处理器。可以实现多处理器（超标量）及相关多级缓存。这就使我们的限制只有内存访问时间及其带宽了。

这些做法造成的结果就是设计复杂性明显更大了（见图 1.23）。我们的高级处理器能有数百万晶体管而不是 100 000 晶体管级的处理器。多级缓存也复杂了，这

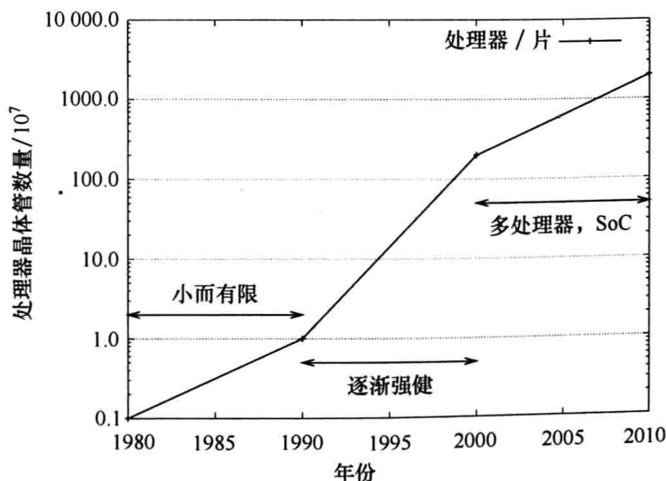


图 1.23　设计复杂性

是为了协调（同步）多个处理器的需要，因为这些处理器要求内存内容是一致的。

为了管理这些复杂性，重用是一种显而易见的方法。所以，相比于一个全新的更高级的单个处理器，在一个芯片上重用几个简单处理器设计会更好。如果能够选择特定的处理器设计来适应应用的特殊部分，上述方法就非常正确了。为了让这种方法起作用，也需要有力的互联机制来访问各种处理器与内存。

所以，当应用确定后，SoC 构建包括以下四方面：

1）多个（通常是）异构的处理器，每个专门为应用的特定部分而工作。

2）用于部分程序存储的主存（通常）外加 ROM。

3）相对简单的小型（单一级别）缓存结构或是与每个处理器都相关的缓冲器。

4）用于通信的总线或开关机制。

尽管 SoC 构建方法在技术上很有吸引力，但是它存在经济上的限制和影响。考虑到处理器及其互联的复杂性，如果将实现的有用性局限于特定应用的话，要么（1）确保这个产品能有很大的市场；要么（2）通过设计重用或是相似技术找到能减少设计成本的方法。

1.9 SoC 产品经济及影响

1.9.1 影响产品成本的因素

产品的基本成本与收益取决于很多因素：技术需求、成本、市场大小及产品在未来产品市场的影响力。在生产成本之外还有些其他成本问题。

成本包括固定成本与可变成本，如图 1.24 所示。事实上，工程成本通常是最大的固定成本，并且其在通过销售获得收入之前就得开销（见图 1.25）。

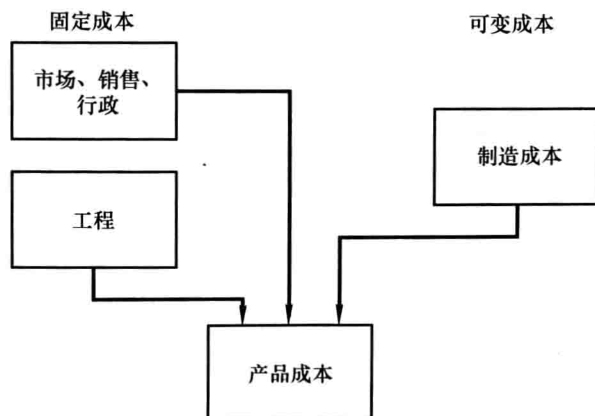


图 1.24 项目成本组成

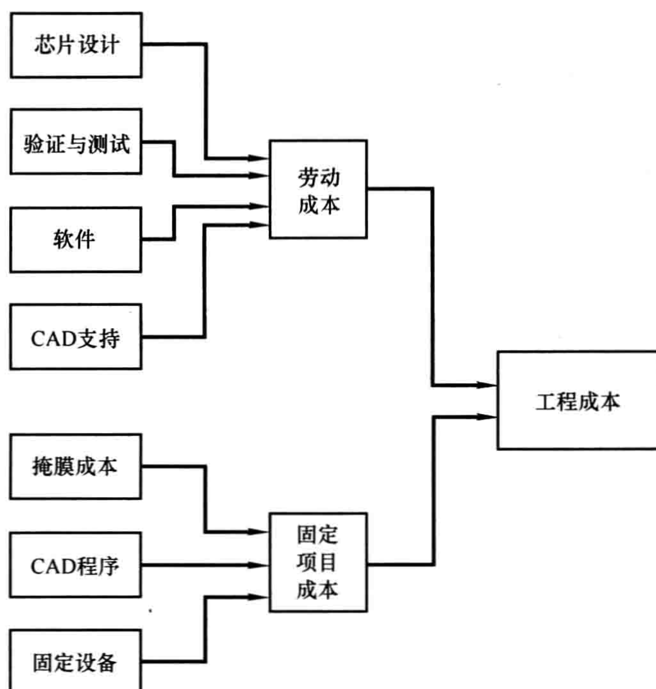


图 1.25　工程（开发）成本

在生产的第一个产品完成之前，设计一个新芯片无论如何都需要 12 ~ 30 个月的开发工作，这还要取决于其复杂性。即使是一个中等规模的项目也需要多达 30 位甚至更多的软硬件工程师、CAD 设计与辅助人员。例如，一篇论文中写道日本索尼公司的情感引擎项目中就有 22 个专家[147,187]。然而，他们的薪水及间接成本只代表了总开发成本的一小部分。

非工程固定成本包括制造启动成本、库存成本、初期市场与销售成本及行政费用。市场成本包括各种显而易见的事项，如市场调研、市场战略规划、定价研究、竞争分析等，当然还有销售计划与广告成本。一般行政（General and Administrative，G&A）“费用”的概念包括成比例的“前台”份额及其他成本，而“前台”是指行政管理部门、人事部门（人力资源）、财务部门。

其后，在生产过程的开始阶段，部件费用仍然很高。边际生产成本只有在生产大量部件之后才会接近最终的制造成本。

再之后，产品部件制造成本会逐渐接近最终的制造成本。在此期间，仍然需要持续不断的开发工作来延长产品周期并拓宽其市场适应性。

所生产的产品将会产生利润吗？从原先的讨论中很容易就可看出成本是如何地敏感于产品周期及产品数量的。如果市场规律或是市场竞争相当激烈并催生性能扩充的竞争系统的话，产品周期就会缩短并导致所交付的产品少于预期。这样一来就会导致灾难性的结果，即使最终的生产成本能控制住。因为将没有足够的

产品来均摊固定成本以确保盈利。而另一方面，如果竞争不激烈，那么第二代产品的开发团队就能够成功地强化产品并维系其市场影响力，然后产品就能成为公司的家底宝贝之一，并给设计者和股东带来名声和微笑。

1.9.2 给产品经济和技术复杂性建模：SoC 课程

为了全盘考虑这些成本，来考量一个通用的模型——产品平均单位成本（这区别于其最终生产成本）：

单位成本 = （项目成本）/（单位数）

项目成本仅是所有固定成本与可变成本的总和。用一个常量 K_f 来表示固定成本。可变成本为 $K_v \times n$，其中 n 是单位数。然而，工程成本、销售成本及市场成本与 n 相关却必然不是线性的。

那么用这样一项来表示这种效果：0.1 倍的 K_f 并再用 n 缓慢地增加它，如 $\sqrt[3]{n}$。这样得到

$$产品成本 = K_f + 0.1 \times K_f \times \sqrt[3]{n} + K_v \times n \quad (1.1)$$

那么，用式（1.1）来说明产品设计中先进技术的影响。将一个 1995 年完成的设计与一个 2005 年的产品成本更低的复杂设计来比较。将 K_f 固定，如图 1.26 所示，1995 年随着产品产量 n 增加，单位成本获得了预期的减少。但是该图也显示出如果扩大固定成本（更复杂的设计）10 倍，即使将单位成本 K_v 削减相同的量，2005 年的单位产品成本直到达到很大的产量时仍旧很高。这种现象对市场巨大的“通用”处理器设计来说或许不是一个麻烦，但对面向特殊应用而产量有限的 SoC 设计来说就会成为一个挑战。对于特殊应用来说，越有针对性的设计就越高效，而这是以牺牲通用性为代价的，当然也就会影响产量。

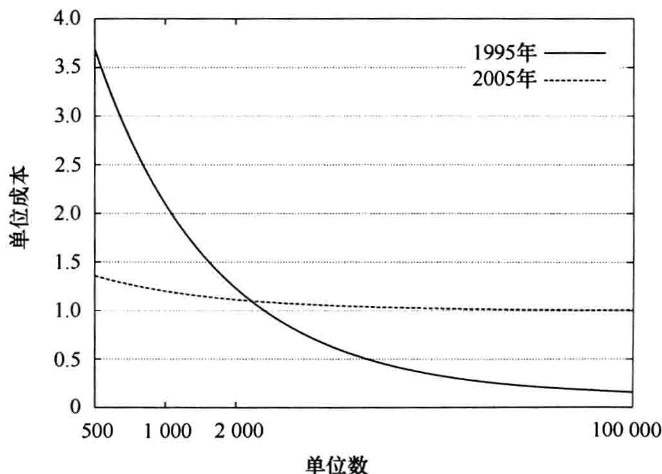


图 1.26 1995 年与 2005 年产品对单位成本的影响

1.10 应对设计复杂性

随着设计成本与复杂性增加，物理产品的设计最优化与设计成本之间存在一

个基本的折中，如图 1.27 所示。这个平衡点取决于 n，即预期要生产的单位数。针对设计生产力问题存在着好几种处理方法。最基本的方法就是购买预先设计的组件及利用可重构设备。

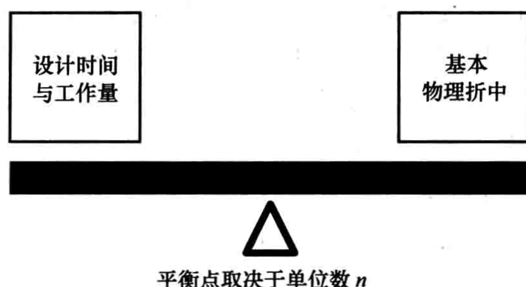


图 1.27　设计工作量必须与产量平衡

1.10.1　购买 IP

如果设计目标是产生一个技术优化的设计，那么固定成本就会很高，所以其成果必须具备广泛的适用性。而另一种方法就是“重用”现有设计。这样做在特殊处理技术的细微差别方面可能是次优的，但是却能显著地节省设计时间和工作量。这种向第三方购买此类设计被称为 IP 销售。

IP 的使用减少了设计开发的风险：它能减少设计成本并缩短产品上市时间。IP 的成本通常取决于容量。因此，采用 IP 的方法是在减少式（1.1）中 K_f 的同时以增加 K_v 为代价的。

专业的 SoC 设计总会使用多种不同类型的处理器。非关键性的及专业的处理器是作为 IP 购买的，并集成到自己的设计中。例如，ARM7TDMA 就是一款通用授权的 32 位处理器或称之为“核心”设计。一般来说，处理器核心能以各种方式用于设计和授权，如表 1.16 所示。

表 1.16　可用处理器核心 IP 类型

设计类型	设计级别	描　述
定制硬 IP	物理级	用于固定的工艺，最优的
综合固 IP	门级	用于很多工艺，有一定的优化
可综合软 IP	寄存器晶体管级（Register Transfer Level，RTL）	用于任何工艺，非最优

硬 IP 是使用处理器技术中所有可用特性的物理级设计，包括电路设计及物理布局。很多模拟 IP 及混合信号 IP（如 SRAM、阶段锁相环）都是以这种形式发布的，这样能确保最优的时序及其他设计特点。固 IP 是门级设计，虽给设备定了尺寸，但对很多采用不同处理器技术的实验室设备来说很合适。软 IP 是逻

辑级设计，具有可综合的格式，能立即适用于标准芯片技术。这种方法允许用户改编源代码以使其设计适应各种情形。

很显然，厂商的设计越优化，通常用户的可定制性就越差，但是这样的设计却经常能获得很好的物理成本-性能折中。在定制过程中，由于设计步骤甚至是产品技术本身需要支持用户定制，因此存在潜在的性能-成本-功耗开销。此外，定制设计也需要再验证以确保其正确性。当前的技术，如接下来描述的重构技术，都是致力于最大化后期定制的优点，如减少风险及缩短上市时间。同时，它们还将与之相关的缺点最小化，如通过引进电路的非可编程的块来支持如整数乘法之类的常见操作，这种电路块比可重构资源更高效，但是却不如其灵活。

1.10.2 重构

重构这个词是指通过很多方法来让同一块电路能在很多应用中重用。可重构设备也可被看成是一种购买的 IP 类型，其制造成本和风险都消除了，然而其在支持用户定制方面会提高单位成本。换句话说，采用可重构设备是在减少式（1.1）中 K_f的同时以增加 K_v为代价的。

这种方法中最著名的例子当属 FPGA 技术了。一个 FPGA 由大量的单元阵列组成。每个单元包含一个小查找表、一个触发器，还可能有个输出选择器。这些单元由可编程连线连接在一起，使得阵列之间具有灵活的路由通路（见图 1.28）。任何逻辑功能都可以通过在 FPGA 上配置查找表与互联线来实现。因为一个阵列能由 100 000 个单元组成，所以很容易就能定义一个处理器。实现这种基于 FPGA 的软处理器的一个明显缺点就在性能-成本-功耗方面。当然，这种方法还是有以下很多优点的：

1）电路制造成本随着时间呈指数级增长。因此，除非大量制造的芯片，否则就不划算。而 FPGA 本身是通用设备，因而被期望可以大量制造。

2）相比于设计一个用于加工的芯片，FPGA 实现的设计时间较短。FPGA 实现有很多可用的设计扩展库。在需要缩短上市时间相当关键的时代，这样做对于设计尤为重要。

3）FPGA 可以快速搭建要制

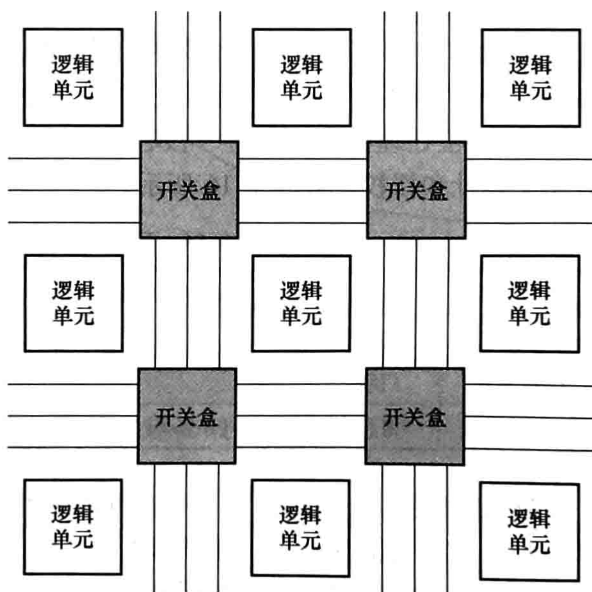


图 1.28　FPGA

造的电路原型。在这种方法中，根据设计计划将一个或多个 FPGA 配置起来模拟它，这是一种“在线仿真”形式。在其上运行程序然后就可以检测到设计错误了。

4）FPGA 的可重构性能方便系统升级，这将有助于提高产品的市场存活时间。这种能力对于那些新功能或新标准快速涌现的应用来说尤为有价值。

5）FPGA 能通过配置先适应部分任务，然后再配置来适应其余任务（称为“运行时配置”）。这使得用于特定运算的特殊功能部件能适应环境变化。

6）在很多计算敏感的应用中，FPGA 能通过配置成为高效的脉动运算阵列。由于每个 FPGA 单元有一个或多个存储元件，能以合适的粒度进行流水线运算。这样能提供一个巨大的运算带宽，在选定的应用上能产生极大的加速。某些设备[25]就是将常规处理器与 FPGA 结合起来的，如 Stretch S5 软件配置的处理器。

在高效 SoC 设计中，重构与 FPGA 扮演了很重要的角色。本书第 2 章将深入探讨这些内容。

1.11　总结

构建现代处理器或目标应用系统是一项复杂的事业。将数百万晶体管集成在单个芯片上的技术在带来很大优点的同时也带了代价，这种代价不是硅本身而是为了实现和支持产品而需要做大量的设计工作。

SoC 设计存在很多方面的内容，如高级描述、编译技术及本章没有提到的设计流程。这些内容以后会涉及。

在接下来的章节，首先将进一步分析技术方面的基本权衡点：时间、面积、功耗及可重构能力；然后，将分析组成系统的组件的一些细节，这些组件包括处理器、缓存、存储器及将它们互联的总线或开关；接着，从用户定制和可配置性的角度来考虑设计与实现方面的问题；紧随其后的是关于 SoC 设计流程及应用研究的讨论；最后，将会介绍未来 SoC 技术方面所要面对的一些挑战。

本书的目标在于帮助系统设计师辨别最高效的设计选择及通过挖掘技术优势来管理设计复杂性的机制。

1.12　习题

1. 假设图 1.18 所示的 TLB 有 256 项（直接寻址）。如果虚拟地址是 32bit，实际内存有 512MB，页大小是 4KB，画出一个 TLB 项的可能布局。图 1.18 所示的用户 ID 的作用是什么？如果忽略它会有什么后果？

2. 讨论进行 TLB 地址转换时可能有哪些安排。

3. 查找一个真实的 VLIW 指令格式。描述在使用应用时单条指令的格式及其对程序的约束。

4. 查找一条真实的用于矢量 ADD 的矢量指令。描述该指令的格式。反复进行向量加载和矢量存储操作。试问矢量指令是否允许重叠执行？并解释原因。

5. 对于图 1.9 所示的流水线处理器，假设指令#3 在 WB 结束时将控制代码（Condition Code，CC。能被其后的分支指令用于测试）置位，且指令#4 是条件分支指令。在没有额外硬件支持的条件下，试问分支成功与分支失败情况下执行指令#5 时的延迟分别是多少？

6. 假设有四个不同的处理器，每个都处理某个应用的 25%。如果将其中两个处理器速度提高 10 倍，那么总体应用加速是多少？

7. 假设有四个不同的处理器，但是除了某一个以外其他处理器都受限于总线。如果将总线速度提高 3 倍，并且假设处理器性能也同比例提高，那么整个应用的加速是多少？

8. 对于图 1.19 所示的流水线处理器，假设每条指令的缓存失效率是 0.05 且失效延迟是 20 个时钟周期。那么 CPI 是多少？在此忽略其他延迟，如分支延迟。

9. 设计验证是 SoC 设计中一个很重要的方面。查找几种专门针对 SoC 设计的验证方法。并从小型 SoC 厂商的角度评价这些方法。

10. 查找（如从网上）两种最新的 VLIW DSP。判别每个时钟周期发射的最大操作数及操作的组成（整数操作数目、浮点操作数目、分支操作数目等）。还有规定的最高性能（每秒指令数）是多少？这个数是怎么计算出来的？

11. 查找（如从网上）两种最新的大型 FPGA 部件。判别其逻辑块数目（即 CLB[⊖]）、最小时钟周期及允许的最大功耗。还有其支持什么样的软处理器？

⊖ CLB：Configurable Logic Block，可配置逻辑块。

第2章　芯片基础：时间、面积、功耗、可靠性和可配置性

2.1　引言

对于任何系统设计，权衡成本和性能都是最基本的。不同的设计，源于对性价比的不同选择，或者对成本和性能本质的不同假设。

设计创新的驱动力在于飞速进步的工艺技术。美国半导体行业协会（Semiconductor Industry Association，SIA）定期做出技术前沿的规划，名为 SIA 蓝图，这越来越成为了新的芯片设计的基础和前提。当 SIA 蓝图做出改变，达到并保持芯片的先进性就变得越来越艰难。表 2.1 给出的 SIA 蓝图是对特定年份的高性能微处理器规划指南的总结[132]。随着光照技术水平的提高，晶体管越来越小。加工工艺限定了晶体管门的最小宽度。表 2.1 用纳米（nm）来指代加工工艺；而老几代的加工工艺往往用微米（μm）来描述。往前几代分别是 65nm、90nm、0.13μm 和 0.18μm。

表 2.1　SIA 蓝图

年　　份	2010	2013	2016
工艺/nm	45	32	22
晶圆尺寸，直径/cm	30	45	45
缺陷密度/(个/cm^2)	0.14	0.14	0.14
微处理器芯片尺寸/cm^2	1.9	2.6	2.6
芯片频率/GHz	5.9	7.3	9.2
晶体管数密度/(百万个/cm^2)	1203	3403	6806
最高性能下最大功耗/W	146	149	130

2.1.1　设计的权衡

随着芯片频率的增长，尤其是晶体管密度的增长，设计者必须能够在快速变化的工艺变化环境中找到最好的权衡。而由于功耗需求，芯片频率已经成为一个不得不考虑的问题。

在做最基本的芯片权衡时，有五点需要考虑。第一是时间，包括事务或指令周期的划分、用于加速执行指令的基本流水线机制和优化程序执行的参数周期。第二个是面积。由于特定功能而导致的面积开销，是权衡架构的另外一个重要方面。第三是功耗。功耗既影响性能也影响实现。需要小面积实现的指令系统比需要更大面积来实现的指令系统有价值的多——虽然后者当然会有与之相匹配的更好的性能。可保持的性价比是大部分设计决策的基础。第四是可靠性，在处理深亚微米影响时，可靠性开始变的重要。第五是可配置性。可配置性为设计者在权衡非一次性设计和一次性设计费用方面提供了一个机会。

SoC 设计的五大问题

其中的四个问题是很明显的。芯片面积（制造成本）和性能（深受周期的影响）是基础 SoC 设计考虑的重要方面。功耗作为一个设计约束也开始崭露头角。工艺缩减了特征尺寸，所以稳定性也将会成为设计考虑的主要问题。

第五个问题是可配置性，它并不是一个非常明显而需要立即就去思考的问题。然而，就像本书第 1 章所讲的，在 SoC 设计中，一次性设计费用是能够主导整个项目成本的。对于 SoC 设计来讲，通过可配置性来做出灵活的设计是非常重要的。这样，既可以扩大市场，又可以降低每一部分的成本。

可编程性支持现场编程，并且提供了“应用定制的制造标准化”特征。集成电路的标准化和定制化之间的可复用的性质是由 Makimoto 发现的[163]，众所周知的 Makimoto 曲线，如图 2.1 所示。

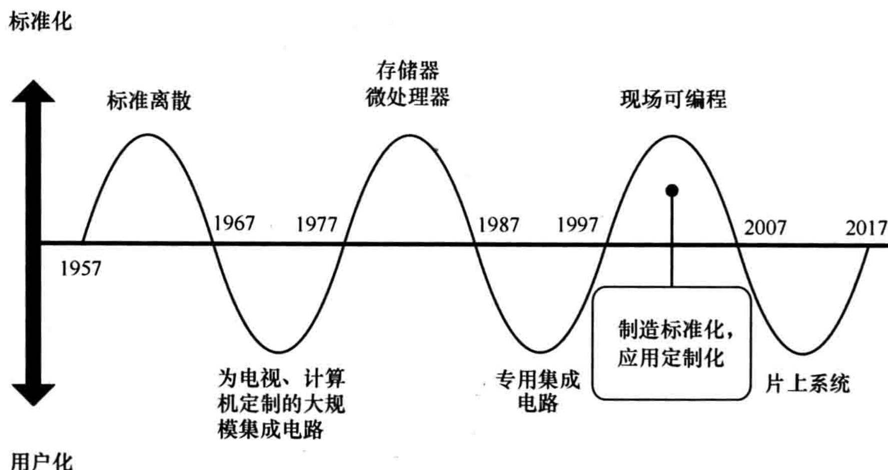


图 2.1 Makimoto 曲线

关于复杂度，有很多种性价比权衡。例如，当特征尺寸固定时，面积可以根据性能（用执行时间 T 来表示）来权衡。大规模集成电路复杂度的理论表明，对于处理器设计存在一个 $A \times T^n$ 的边界范围，并且 n 往往落在 $1 \sim 2$[244]。也可以

用边界范围 $P \times T^3$ 来权衡时间 T 和功率 P。处理器设计权衡如图 2.2 所示，在处理器设计过程中可能的权衡方面包括面积、时间和功率[98]。在这个三维空间中，嵌入式处理器和高端处理器在不同的设计域中得以实现。功率-面积轴线主要用于嵌入式处理器的优化，同时时间轴线主要用于高端处理器。

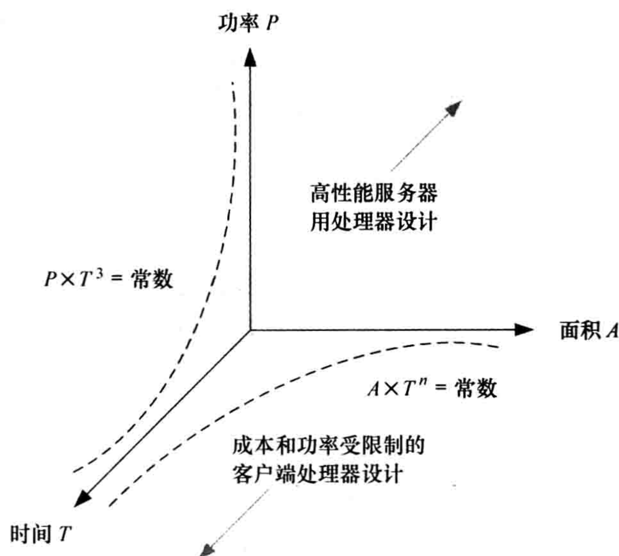


图 2.2 处理器设计权衡

本章主要解决在设计中需要权衡的问题。首先是时间问题。性能的最根本衡量标准是完成系统任务和功能所需要的时间。这取决于两点：一是处理器存储器的组织及大小，二是它们的基本频率或者时钟频率。第一个问题将在本书第 3、4 章解决。本章只看基本的处理器周期——一个周期中发生了多少延时，指令在执行过程中是怎样被划分成周期的。接下来介绍一个成本（面积）模型来协助权衡制造过程中的成本。这个模型仅限于片上和处理器类型的权衡，但是它诠释了一种系统设计的模型。正如本书第 1 章所提到的，硅片成本通常是总成本中很小的一部分，但是对它的了解仍然是很有必要的。功耗一般是由周期时间和设计的总尺寸及它的器件所决定的。在很多 SoC 设计中，它已经成为了主要限制。最后，考虑可靠性和可配置性及它们对成本和功能的影响。

2.1.2 需求和规格

五个基本的 SoC 权衡项为分析 SoC 需求提供了一个框架，它可以转化成规格。成本需求和市场规模可以转化成芯片成本和工艺。耐用性和重量限制的需要会影响热量损耗，可以转化成功耗范围和时钟周期限制。对于一个特定的设计，任何一个权衡标准都有一个最高优先级。看以下几个例子：

- 高性能系统将以成本和功率（甚至可能也有可配置性）作为代价使得时间最优化。
- 低成本系统优化芯片的成本、可配置性和设计的重用性（也可以有低功耗）。
- 可携带系统强调低功耗，因为能量提供决定了系统的重量。所以这样的系统如手机等，通常有实时性的限制，而功能可以忽略。

- 应用在飞机或者其他安全主导的嵌入式系统强调可靠性，功能和适用寿命（可配置性）是第二个重要的考虑因素。
- 游戏系统强调成本——尤其是产品成本，第二是性能，对可靠性的考虑很少。

在需求方面，SoC 设计者需要认真地考虑每一个权衡项，并确定与之相一致的规格。本章会将 SoC 需求因素转换成相应规格并开始学习优化设计选项，同时也会有对系统组成的本质理解，这些在后面章节也会看到。

2.2 周期

时间的概念受到了处理器设计者的高度关注。它是性能的最基本度量标准；然而，将行为分解成周期并且降低周期数目和周期时间很重要，但并非是精确的科学。

行为分解成周期的方式是很重要的。一个常见问题就是一个基本行为需要意料之外的“额外”周期，如缓存不命中。总体而言，对于周期选择和将指令执行分解成为周期的理论基础有限。很多设计都是基于语用的。

在这一部分，将介绍一些指令划分的技术，也就是将指令执行时间划分为可控和固定时间的技术。在流水线处理器中，数据流过每一阶段和项目流过装配线几乎一样。在每一阶段的结束，一个结果传递给后面一阶段，同时一个新的数据流入进来。在一定限制之内，周期越短，流水线就越高效。然而，分解的过程是有开销的，而且最小周期时间是受开销控制的。简单的周期模型可以优化流水的阶段数目。

流水线处理器

曾经，处理器中流水线概念被认为是一种先进的处理器设计技术。的确，在过去的几十年里，流水线已经成为了任何一个处理器和控制器的主要部分。它是在决定处理器或系统的周期和执行时一个最基本的考虑方面。

周期和流水阶段数目之间的权衡将会在流水线优化部分探讨。

2.2.1 周期的定义

周期（时钟的）是处理信息的基本时间单元。在同步系统中，时钟频率的值是固定的，周期时间是由机器中频繁发生操作的最大完成时间决定的，如加法器或寄存器数的传递。这个时间必须足够用于一个数据存到一个特定目的寄存器（见图 2.3）。小频率操作需要更多的时间和周期来完成。

一个周期起始于指令译码器（基于当前的指令操作码）为系统寄存器指定一个数值。这些控制数值将一个特定寄存器的输出与另一寄存器或者加法器或者



图 2.3　一周期可能的操作队列

相似的目标联系起来。这就使得来自于源寄存器的数据通过指定的组合逻辑传递到目标寄存器。最后，经过一段合适的建立时间之后，所有的寄存器通过时钟系统的边沿或脉冲进行采样。

在同步系统中，周期时间是由周期内最坏情况下的每一步或动作的时间之和决定的。然而，时钟本身可能不会在预期的时间到来（由于传输或加载的影响）。偏离期望时钟到来的最大时间称为时钟歪斜。

在异步系统中，周期时间是简单的由事件或操作的完成来决定的。一个完整信号生成之后，才允许下一个操作开始。由于完成信号开销和流水时序约束，异步设计通常并不用于流水线处理器。

2.2.2　流水线优化

对于流水线处理器设计者来讲，一个最基本的优化就是将流水线并行分解。分解的数量越大，允许的最大速度就越高。然而，每个新的分割都会带来时钟开销，这将对性能有不利的影响。

如果忽略行为分割与整数个周期匹配的问题，可以确定一个优化周期时间 Δt 和之后简单流水线处理器的分割层次。

假设没有进行流水分割时执行一条指令的总时间是 T（单位为 ns）（见图 2.4a）。那么问题是考虑时钟和流水，找到最优的分割数目 S。通过一次分割的理想延时是 $T/S = T_{seg}$。与每个分割相关的是分割开销。这个时钟开销时间（纳秒级）包括时钟歪斜和任意一个寄存器所需的建立和保持信号的时间。

那么，流水线处理器的真正周期时间（见图 2.4c）就是理想的周期时间 T/S 加上时钟开销。

$$\Delta t = \frac{T}{S} + C$$

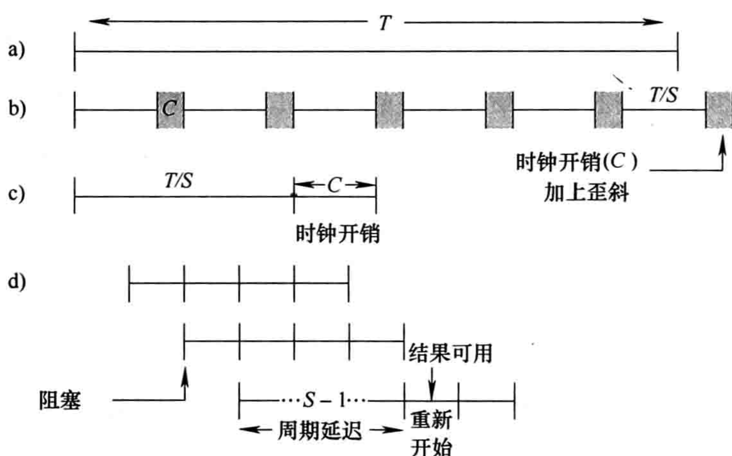


图 2.4 最优流水线

a）无时钟指令执行时间 T b）T 被分割成 S 段（每段需要时钟开销 C）

c）时钟开销和它对周期时间的影响 T/S d）流水组涩的影响（也称流水中断）

在理想化的流水线处理器中，如果没有码延迟，指令是以每周期一条的频率进行的，但延迟是会发生的（主要是由于猜测错误和不期望的分支造成的）。假设这些中断发生的概率是 b 并导致 $S-1$ 条正准备进入或者已经进入流水线的指令失效（图 2.4 给出了一个最坏情况的阻塞）。流水中断有很多不同种类，有不同的影响，但是这个简化模型诠释了中断对性能的影响。

考虑流水中断，处理器性能为

$$性能=\frac{1}{1+(S-1)b}指令/周期$$

吞吐量 G 可以定义为

$$G=\frac{性能}{\Delta t}指令/\text{ns}$$

$$=\left(\frac{1}{1+(S-1)b}\right)\times\left(\frac{1}{\frac{T}{S}+C}\right)$$

如果找到 S，使得

$$\frac{\mathrm{d}G}{\mathrm{d}S}=0$$

那么就找到了 S_{opt}，最优的流水分割数目为

$$S_{opt}=\sqrt{\frac{(1-b)T}{bC}}$$

一旦初始 S 确定，总指令执行延迟（T_{instr}）为

$$T_{instr}=T+S\times(时钟开销)=T+SC$$

$$或\ S(T_{seg}+S)=S\Delta t$$

最终，计算出用每秒钟执行百万指令数来表示的吞吐性能。

假设 $T=12.0\text{ns}$，$b=0.2$，$C=0.5\text{ns}$。那么 $S_{opt}=10$ 段。

如上确定的这个 S_{opt} 是过分简化了的，并没有具体考虑函数单元不能被随意分割，必须遵守整数周期边界等。尽管如此，确定 S_{opt} 可以起到设计起始点的作用，也可以在其他经验性的优化设计方面作为重要的检查。

在此之前的讨论中，只考虑基于性能基础上的流水段数目 S。每当新的流水段被引进，就会增加额外的成本，这并没有作为分析的因素。每个新的流水段都需要额外的寄存器和时钟硬件。因此，最优化流水段数目 S_{opt} 理应被理解为一个特定处理器能够使用的流水段数目可能的上限值。

2.2.3 性能

较高的时钟频率加上较少的流水段不一定能够产生更好的性能。的确，考虑到延时缩放问题，会立即问到，规划的时钟频率是如何得到的。有两个基本要素可以提高时钟频率：①提高对时钟开销的控制；②在流水线种增加流水段数目。图 2.5 给出了一个周期内门延迟，说明了流水段的门延迟已经显著减少了约 4/5，数据源于单元的标准门延迟。这个标准门有一个输入并且驱动了四个相似的门作为输出。这个延迟叫做四扇出（Fan-Out of Four，FO4）门延迟。

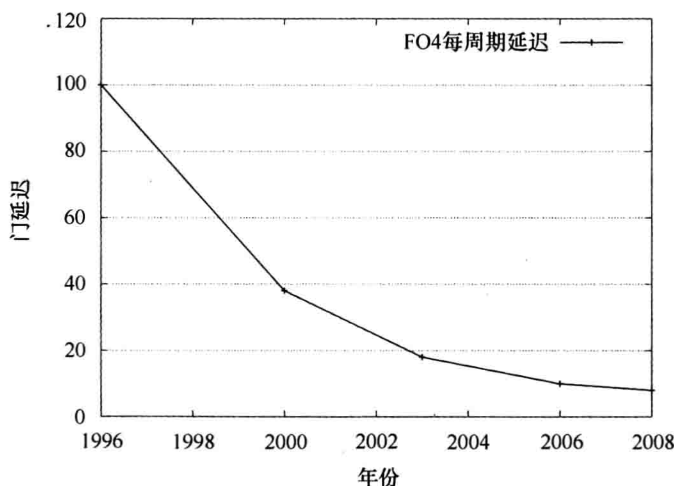


图 2.5 一个周期内门延迟（FO4）

低时钟开销（small C）可能会减小流水段长度，但性能并不一定相应的提高，除非也降低流水阻塞率 b。为了达到高时钟频率，处理器也需要使用大量的分支表缓冲区和分支矢量预测表，通过这些分支来显著降低延迟。然而，阻塞也

可能来自于缓存不命中，这就需要另外一种方案：多层次、超大容量的片上缓存。通常这些缓存结构占了芯片面积的 80% ~90%。在提高性能方面，底部的处理器远没有提高缓存存储系统的效率重要。

2.3 芯片面积和成本

周期时间、机器结构和存储配置决定了机器的性能。与确定总成本相比，确定性能相对更直接一些。

好的设计，是在特定目标性能基础之上得到最优化的性价权衡的。这也决定了处理器设计的质量。

这里，就来看看制造一个系统时由芯片面积因素决定的边际成本。当然，系统设计者也必须意识到芯片面积的固定成本和其他不同的成本所带来的明显副作用。例如，设计复杂度的显著上升会直接影响它的服务能力、文件编制成本（Documentation Cost）或者硬件走向市场所需要的精力和成本。即使无法准确量化它们的影响程度，但这些影响因素必须牢记于心。

2.3.1 处理器面积

SoC 的尺寸一般在每边 10 ~ 15mm。芯片是从一个更大的直径 30cm（约 12in[⊖]）左右的晶片上做出来的。表面看上去似乎可以简单地通过扩大芯片面积，使得每个晶圆上产生更少的芯片，那么这些更大的芯片就可以很容易地容纳设计者所希望的任何功能。不幸的是，晶圆和工艺都没有达到那么高的水平。在晶圆表面，缺陷随机发生（见图 2.6）。大的芯片要求在其区域范围内不能有缺陷。如果芯片相对于特定工艺面积太大，那么成品率就会很小或者为零。图 2.7 给出了成品率和芯片面积的关系。

好的设计并不需要有最大的成品率。将设计面积降低到某个特定值仅会对成品率有边际影响。另外，小的设计会浪费面积，因为引脚和在晶片上与其他邻近芯片的隔离本身就需要面积。

设计者可用的面积是一个跟制造工艺相关的函数。这包括没有灰尘和其他杂质的纯硅晶体和工艺的整体控制。提高工艺水平可以使在更大芯片切块时达到更高的成品率。虽然光刻技术和工艺都有提高，但是涉及的参数并没有相应的缩放。成功的设计者必须对预期的工艺发展有足够的前瞻性，这样即使早期的设计成品率很低，但随着工艺的进步，设计周期得以延长，成品率有所提高，这样就可以通过广泛的基础产品来摊销设计团队的固定成本。

⊖ in：英寸，1in = 2.54cm。

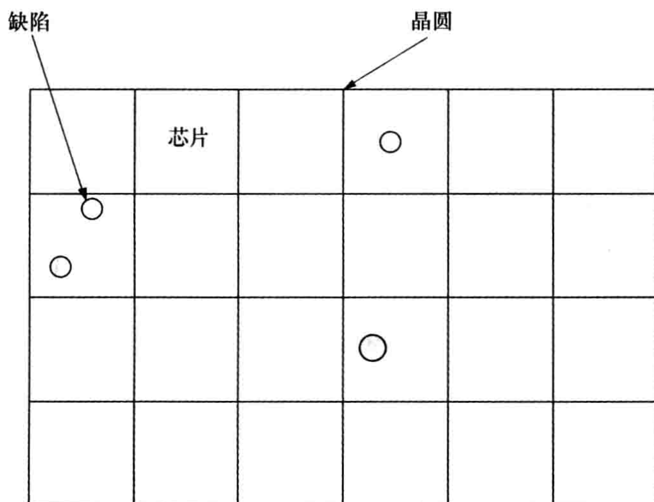


图2.6　晶圆上的缺陷分部

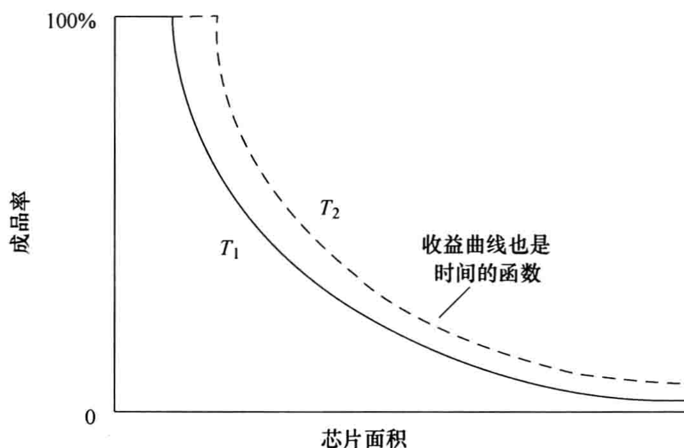


图2.7　成品率和芯片面积的关系

假设矩形的芯片面积为 A。在一个直径为 d 的晶圆上大概有 N 个芯片块（见图2.8），有

$$N \approx \frac{\pi}{4A}(d-\sqrt{A})^2$$

这样修正直径的晶圆被芯片分割。现在假设在晶片上有 N_G 个好的芯片和 N_D 个缺陷点。即使 $N_D > N$，也可能得到几个好的芯片，因为缺陷是随机分布的，几个缺陷点可能聚集在一个缺陷芯片上，浪费了少量芯片。

根据 Ghandi 的分析[164]，假设随机在晶圆上加一个缺陷；N_D/N 就是这个缺陷毁掉一个好芯片的概率。注意，如果缺陷落在了已经坏掉的芯片上，那么就不会引起好的芯片数量的变化。换句话说，好芯片数 N_G 的变化与缺陷数 N_D 变化的比为

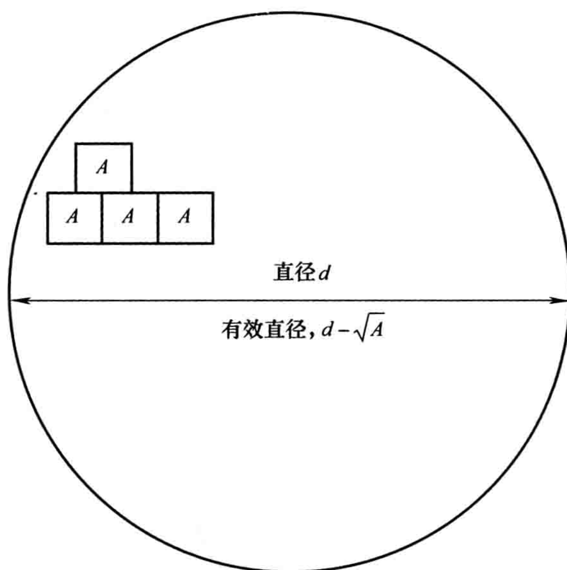


图 2.8　直径为 d 的晶圆上芯片的数量（面积为 A）

$$\frac{\mathrm{d}N_{\mathrm{G}}}{\mathrm{d}N_{\mathrm{D}}} = -\frac{N_{\mathrm{G}}}{N}$$

$$\frac{1}{N_{\mathrm{G}}}\mathrm{d}N_{\mathrm{G}} = -\frac{1}{N}\mathrm{d}N_{\mathrm{D}}$$

积分求解得

$$\ln N_{\mathrm{G}} = -\frac{N_{\mathrm{D}}}{N} + C$$

解 C 的值，注意当 $N_{\mathrm{G}} = N$ 时，$N_{\mathrm{D}} = 0$；所以 C 必定的为 $\ln(N)$。

那么成品率为

$$\frac{N_{\mathrm{G}}}{N} = \mathrm{e}^{-N_{\mathrm{D}}/N}$$

这说明了缺陷是泊松分布的。

如果 ρ_{D} 是每单元面积的缺陷密度，那么有

$$N_{\mathrm{D}} = \rho_{\mathrm{D}} \times (\text{晶圆面积})$$

对于大的晶圆 $d \gg \sqrt{A}$，晶圆直径明显大于芯片边长，故

$$(d-\sqrt{A})^2 \approx d^2$$

$$\frac{N_{\mathrm{D}}}{N} = \rho_{\mathrm{D}} A$$

所以，成品率为

$$\mathrm{e}^{-\rho_{\mathrm{D}} A}$$

图 2.9 给出了不同缺陷浓度下好芯片数与芯片面积的对比，是几种不同缺陷密度下芯片面积与预期成品数的函数关系。工艺的成熟性和设备的费用决定了 ρ_D 的大小，当前，最好的现代化设备的 ρ_D 一般在 0.15 ~0.5 之间。

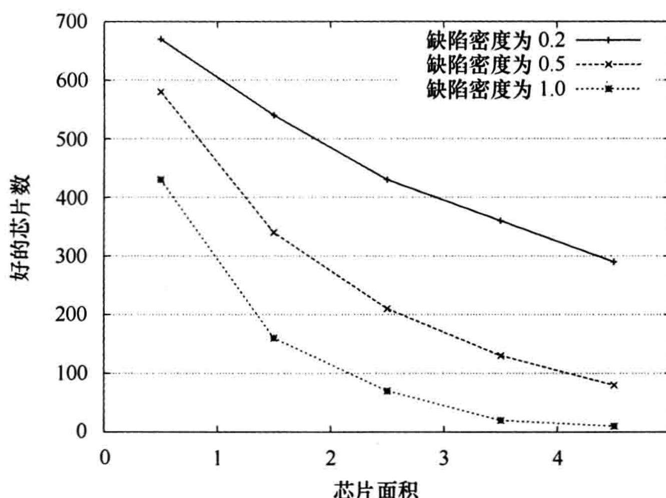


图 2.9　不同缺陷浓度下好芯片数与芯片面积的对比

大尺寸的芯片成本是很高的。对于当前已经很大的 $\rho_D \times A$（约为 5 ~10）来讲，芯片尺寸加倍对成品率的影响是非常显著的。因此，大芯片设计者确信，工艺的发展迟早会降低 ρ_D 来为赚钱的产品提供足够的成品率。

2.3.2　处理器单元

在系统或处理器内部，设计的特定单元所占用的面积是其成本的一个基本衡量标准。在做设计选择或计算一个特定设计的相对优势时，边际效用原则是经常使用到的：假设有一个完整的基础设计和一些扩展设计的额外引脚/面积，选择最佳的引脚扩展面积。在没有输出引脚信息时，假设这个面积是特定的性价权衡的一个主导因素。

很显然面积的单位是平方毫米，但是因为光刻和几何最小特征尺寸是在不断变化的，无量纲单位会更好一些。其中，Mead 和 Conway[164] 用参数 λ 来解决这个问题，它是任意两层掩膜之间的最小几何尺寸。扩散区的最小尺寸为 2λ，与其相邻扩散区间的尺寸可能为 3λ。

假设从一个 $2\lambda \times 2\lambda$ 的晶体管入手，那么标称 $2\lambda \times 2\lambda$ 的晶体管可以扩展为 $4\lambda \times 4\lambda$，至少需要 1λ 来与其他晶体管进行隔离，需要整个晶体管至少需要 $25\lambda^2$ 的面积。这样，单个晶体管的面积是 $4\lambda^2$ 放置在 $25\lambda^2$ 的区域中。

最小特征尺寸 f 是多晶硅门的长度，或者说是一个晶体管的长度 $f = 2\lambda$。很

明显，可以用 λ^2 来定义设计，并且可用任何其他处理器尺寸（门、寄存器大小等）来表示晶体管的数量。所以，面积单位的选择是相对任意的。然而，好的单位是能够代表基本架构的权衡。寄存器等效位（register bit equivalent，rbe）是一个非常有用的单位。它定义了一个六晶体管寄存器单元，表示大概 $2700\lambda^2$ 的面积。很明显它要比单晶体管面积的六倍要大，因为它包含了更大的晶体管、管间的连接和内部位之间必要的隔离面积。

静态 RAM（Static RAM，SRAM）单元具有更小的带宽，面积要比 rbe 小很多，而一个 DRAM 位单元用的面积就更小了。根据经验得到它们之间的关系，见表 2.2。

表 2.2 独立于工艺的相对面积 rbe 和 A 的总结

（只要给出特征尺寸 f，可以算出任何的芯片面积）

面积单位为 rbe	
一个寄存器位	1.0rbe
片上缓存的一个 SRAM 位	0.6rbe
一个 DRAM 位	0.1rbe
rbe 等价于（用特征尺寸为 f）	1rbe = $675f^2$
面积单位为 A	
特征尺寸为 $f = 1\mu m$ 时 A 为 $1mm^2$	
1A 或者约为	$= f^2 \times 10^6$（f 单位为 μm） ≈1481rbe
一个简单的有 32 个字每字 32 位的整数文件（1 读 + 1 读/写）	= 1444rbe
或者约为	≈1A(= 0.975A)
一个 4KB 的直接映射缓存 或者约为	= 23524rbe ≈16A
简单通用缓存（标签和控制位数少于数据位数的 1/5）	= 4A/KB
简单处理器（近似）	
32 位处理器（没有缓存和浮点计算）	= 50A
32 位处理器（没有缓存但是包含 64 位浮点计算）	= 100A
32 位单核并行矢量处理器，没有缓存和矢量存储	= 200A
内部锁存器、总线、控制和时钟单元所需面积	额外需要处理器面积的 50%
Xilinx FPGA	
一个块（2LUT + 2FF + MUX）	= 700rbe
Virtex 4 的一个可配置逻辑模块（4 小块）	= 2800rbe ≈ 1.9A
一个 18KB 的 RAM 模块	= 12600rbe ≈ 8.7A
嵌入式 PPC405 核	≈250A

表中，寄存器堆的面积由寄存器位数和可存取堆的端口 P 数量决定：

$$寄存器堆面积 = (寄存器数 + 3P)(寄存器位数 + 3P)\text{rbe}$$

缓存面积用的是 SRAM 位模型，由缓存位的总数所决定，包括数组、索引和控制位。

芯片或部分芯片上单位 rbe 的数量在迅速增大，所以通常用一个更大的静态单位比较容易。我们简单地将这个单位定为 A，定义在 $f = 1\mu m$ 时其值为 $1mm^2$ 的芯片面积。这同时也是一个 32 × 32 的三端口寄存器堆或者 1481rbe 的面积。

晶体管密度、rbe 和 A 都是特征尺寸二次方的一个扩展。正如表 2.3 所示，对于特征尺寸 f，$1mm^2$ 面积上 A 的数目就是 $(1/f)^2$。与 $1\mu m$ 的特征尺寸相比，在特征尺寸 45nm 工艺下，$1mm^2$ 面积上晶体管的数量 rbe 值的大概是其 500 多倍。

表 2.3　不同特征尺寸的晶体管密度（以 A 计量）

特征尺寸/μm	每 mm^2 的 A 数	特征尺寸/μm	每 mm^2 的 A 数
1.000	1.00	0.090	123.46
0.350	8.16	0.065	236.69
0.130	59.17	0.045	493.93

2.4　理想和实用尺寸

在特征尺寸和晶体管缩小的同时，希望晶体管的密度也以特征尺寸二次方的数量级提高。相似的，传输延时（门延时）应该会与特征尺寸（与电容的降低一致）线性相关地降低。实用尺寸则不同，因为线延时和线密度的尺寸变化率与晶体管尺寸不同。特征尺寸减小的同时线延时几乎保持不变，因为偏移电阻增大，长度和电容减小。图 2.10 说明了线延时相对门延时的主导地位的上升，尤其是在特征尺寸小于 0.10μm 的时候。相似地，当特征尺寸小于 0.2μm 时，晶体管密度的提高就会一定程度上比特征尺寸的二次方小。通常建议认为将 1.5 作为扩大因子更准确，如图 2.11 所示；换句话说，规模是以 $(f_1/f_2)^{1.5}$ 而不是 $(f_1/f_2)^2$ 扩大。在缩放过程中发生的事情是很复杂的。不仅特征尺寸减小了，工艺的其他方面也发生变化，并且通常是提高的。所以，铜线和许多其他的布线层就可以实现，并且提高电路设计。主流工艺的改变会以不连续的方式影响到尺寸。只要设计者能够利用新一代工艺的全部属性，布线的限制带来的影响就会戏剧性地改善。简单的设计尺寸可能紧紧以 1.5 倍指数的速度扩大，但利用新工艺特征的新一代实现方式可以达到 2 倍指数的扩大速度。为了简化起见，后面，我

们将根据上面的理解利用理想尺寸。

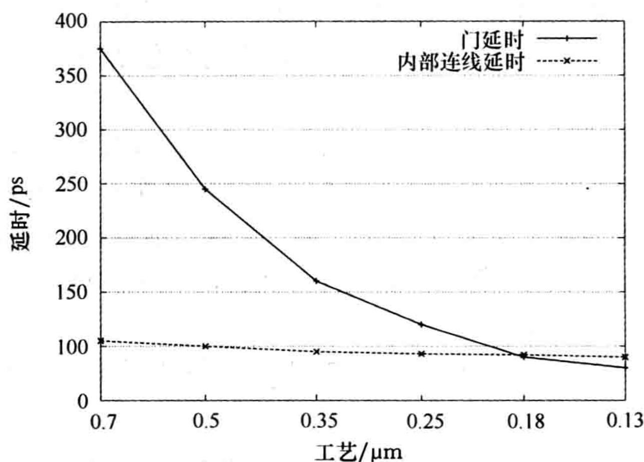


图 2.10　线延时与门延时对比

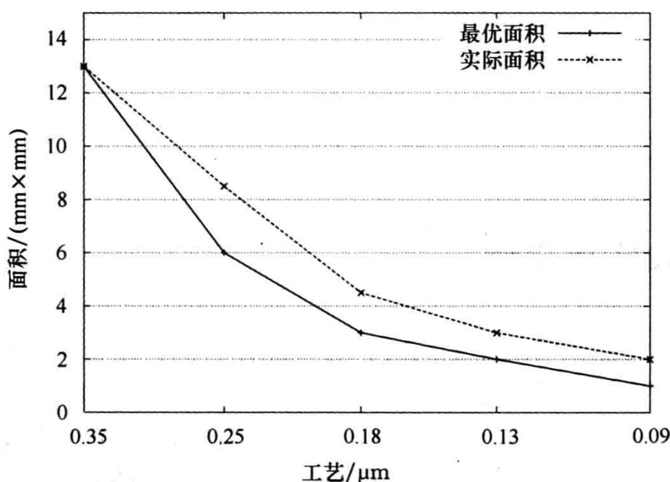


图 2.11　最优和实际缩放的芯片面积

研究 2.1　一个基本的 SoC 面积模型

高效系统设计的关键是芯片层次设计。芯片层次设计的过程跟电阻的层次设计过程没有太大区别。处理器的每个功能部分必须放置在足够大的空间来实现。经常交流的功能单元必须紧密地放在一起。连接路径上必须分配足够的空间。

为了解释在优化芯片层次设计的过程中可能会做出的权衡因素，给大家介绍一个基准系统。在此系统中，特定区域可以完成不同的功能。面积模型是基于经验发现的，这些发现源于现有芯片、设计经验和某些情况下的逻辑推理（如浮

点加法器和整数 ALU 的关系）。这里所描述的芯片在任何意义上都不应该是最佳的，而是当今市场上一定数量设计中的典型例子。

出发点。设计起始于对半导体设计工艺参数的理解。假设能够使用缺陷密度为0.2个缺陷/cm^2的制造工艺；出于经济考虑，目标成品率大约为95%，有

$$Y = e^{-\rho_D A}$$

令 $\rho_D = 0.2$ 个缺陷/cm^2，$Y = 0.95$。则

$$A \text{ 为 } 25mm^2$$

或者说大概为0.25cm^2。

所以可用的芯片面积为25mm^2，这是芯片的总面积，但是像连接芯片与外部世界的线端焊点、这些连接点的驱动和供能电路等都会降低设计者能够利用的芯片面积。假设利用12%的芯片面积（通常是芯片外围）来完成这些功能，那么净面积为22cm^2（见图2.12）。

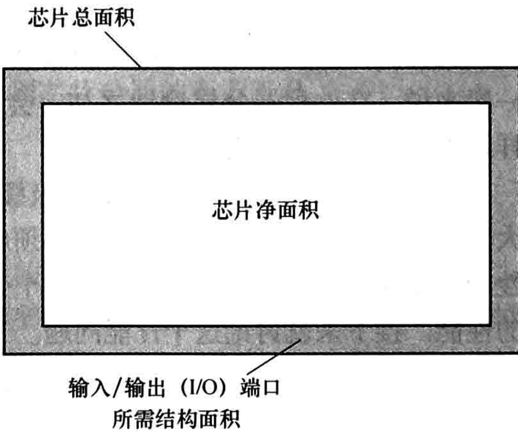


图2.12　芯片净面积

特征尺寸。特征尺寸越小，就有越多的逻辑单元可以在固定的面积内实现。在特征尺寸 $f = 65nm$ 时，可以有大约5200*A*或者22 mm^2的面积单元。

体系结构。根据定义，几乎每个系统都是不同于其他系统的，每个系统都有不同的目标。例如，假设我们需要具备如下结构：

- 一个小的32位处理器核，8KB的指令缓存和16KB的数据缓存。
- 两个32位矢量机，每个矢量机有16个1K × 32b的矢量存储体；一个8KB指令缓存和一个16KB数据缓存来存放标量数据。
- 一个总线控制单元。
- 128KB的实地址应用存储器。
- 一个共享二级缓存。

面积模型。下面是根据系统中不同单元需求对面积的分解。

单　　元	面积/A	单　　元	面积/A
处理器核（32^b）	100	矢量寄存器和缓存2	352
缓存核（24KB）	96	总线和总线控制（50%）	大约小于650
矢量机1	200	应用内存（128KB）	512
矢量寄存器和缓存1	256 + 96	总计	2462
矢量机2	200		

锁存器、总线和内部控制。对于每个功能单元，都有一定的开销，包括非特殊存储器（锁存器）、内部通信（总线）和内部单元控制。其中锁存器的开销占10%，总线、工艺路线、时钟和所有的控制占40%。

系统总面积。专用处理器单元和存储部件占2462A。这样剩下净(5200－2462)A＝2738A 的面积可以用作缓存。注意，芯片是高度的以存储为导向的。剩下的面积将会分配给二级缓存。

缓存面积。缓存能够分配的净面积为2738A。但是对于缓存设计者来说，芯片上零零碎碎没有被占用的面积并不一定有用。这些碎片必须合理地集中在一个紧凑的区域才能够设计出高效的缓存。

例如，一个长宽比很大的可用区域明显比一个紧凑或方形的区域可用性差。一般来说，在处理器分层设计之初，会为长宽比失调分配另外10%的开销。这样剩下可用的净面积就成了2464A。

这样就得到了大约512KB 的二级缓存。这合理吗？在这一点上只能说这么大的缓存对于芯片是合适的。现在必须应用一下并确定这个分配是否会得到最高性能。或许更大的存储器或者另外一个矢量机或者更小的二级缓存能够得到更好的性能。接下来就讨论这个性能问题。

图2.13 给出了一个芯片分层设计的基本框架实例。设计面积的规则总结如下：

1. 根据缺陷密度和成品率的目标，算出目标芯片的尺寸。
2. 算出芯片成本，确定它是否符合要求。

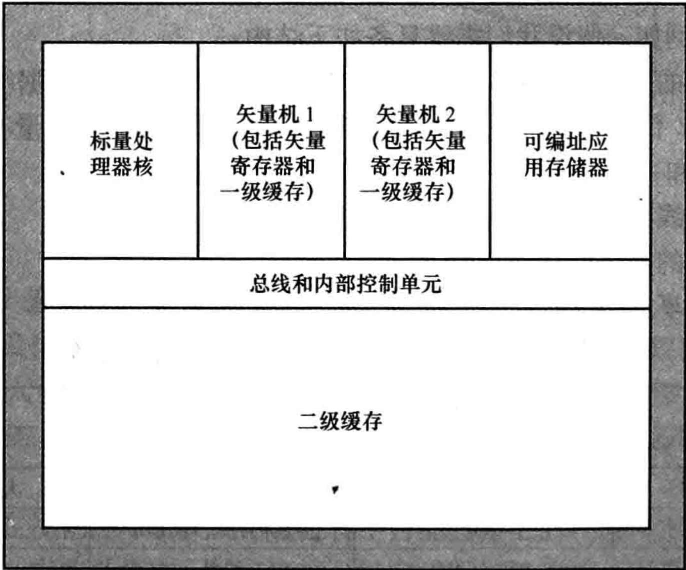


图2.13 一个芯片分层设计的基本框架实例

3. 计算净可用面积。为引脚、保护环、电源等留出 10% ~20% （或其他合适比例）的面积。

4. 根据最小特征尺寸，算出 rbe 的尺寸。

5. 在试验的系统架构基础上分配面积，直到基本的系统面积得以确定。

6. 将 5 中的系统基本尺寸减掉 3 的净可用面积。得到的就是芯片缓存和存储器可用的最优面积。

注意，在这个研究当中（并且对于很多小特征尺寸设计更加确定），芯片的大部分都被分配给了一种或另一种存储器。最基本的处理器面积大约为 20%，其中部分面积包括总线和控制部分。因此，只要对存储器的估计是准确的，那么不论对处理器核和矢量机面积的估计多么粗略都不会影响芯片面积分配的准确度。在特定的设计条件下，有很多用来计划芯片分层的商业工具。

2.5　功耗

近来，对无线和便携式电子产品需求的增加使得更多的关注放在了功耗上。美国 SIA 指出了微处理器芯片由于工作频率的增高、整体电容的增大和尺寸的增大而存在日益增长的功耗问题。功耗是芯片消耗的功率，它并不是由特征尺寸直接决定的，因为它最基本的决定因素是频率。表 2.4 给出了芯片按功耗分类的情况。

表 2.4　芯片按功耗分类的情况[132]

类　型	芯片功率	电源和工作环境
冷却高功耗	70.0W	外插，冷却环境
高功耗	10.0 ~50.0W	外插，风扇
低功耗	0.1 ~2.0W	可充电电池
超低功耗	1.0 ~100.0mW	AA 电池
极低功耗	1.0 ~100.0μW	钮扣电池

在晶体管级，其总功率 P_{total} 包括两个主要部分：动态或切换功率和由漏电流引起的静态功率，即

$$P_{total} = \frac{CV^2 f}{2} + I_{leakage} V$$

式中，C 为晶体管电容；V 为电源电压；f 为晶体管的切换频率；$I_{leakage}$ 为漏电流。直到最近，切换功率一直是总功率的主导因素，但是现在静态功率也在增加。从另外一个角度来讲，门延时大概跟 $CV/(V - V_{th})^2$ 成正比。其中，V_{th} 为晶体管的阈值电压（对于逻辑级开关）。

特征尺寸减小的同时，晶体管尺寸也在减小。更小的晶体管尺寸导致了电容

降低。电容降低的同时降低了动态功率和门延时。随着晶体管尺寸的减小，应用在其中的电场已经大到毁灭性的程度。为了提高晶体管的可靠性，我们需要降低电源电压 V。降低电压 V 有效地降低了动态功率，但是却导致门延时增加。可以通过降低 V_{th} 来避免这种损失。而另一方面，降低 V_{th} 提高了漏电流，因此也就提高了静态功率。这对设计和产品的影响是很大的；在产品中必须有以下两种晶体管设计：

1. 低 V_{th} 值和高静态功率的高速晶体管。
2. 维持 V_{th} 和 V 不变，以电路密度和低静态功率为代价的低速晶体管。

不论哪种情况，都可要通过降低电源电压 V 来减小开关功率。Chen 等人[55]说明了漏电流的正比值为

$$I = (V - V_{th})^{1.25}$$

式中，V 为电源电压。

根据上面的讨论可以看到，信号传递时间和频率随充电电流而变化。所以，最大工作频率也是跟 $(V - V_{th})^{1.25}/V$ 呈正比关系的。对 V 和 V_{th} 的值感兴趣，也就是说频率随电源电压呈比例变化。

假设 V_{th} 是 0.6V；假如将电源电源降为 1/2，也就是从 3.0V 降为 1.5V，那么工作频率也降为了大约原来的 1/2。所以，电源电压减半，工作频率也将减半。

根据功率方程（电压和频率都已经减半），总功耗变为了原来的 1/8。因此，如果对现存设计的频率进行优化并且将其改为低电压运行，那么频率将会大概降为最初静态功率的三次方根：

$$\frac{f_1}{f_2} = \sqrt[3]{\frac{P_2}{P_1}}$$

理解对现存设计的频率的缩放和功耗优化实现之间的区别是非常重要的。功耗优化的实现和功能优化的实现在几个方面是不同的。

功率优化的实现是用更少的芯片面积，不仅是因为要降低供电和时钟分配的需求，而且更重要的是因为性能目标的降低。性能主导的设计利用了大量的面积使到性能的边际提高，如用超大规模的浮点单元、最小时钟倾斜的分布网络和最大尺寸的缓存。但对于利用电池的便携式或者无线处理器来讲，功率是最关键的问题，而不是性能。表 2.5 给出了一些电池的容量和工作周期。

表 2.5　一些电池的容量和工作周期

电池类型	电量/(mA·h)	工作周期、寿命	功　　率
可充电电池	10000	50 小时（10% ~20% 使用时间）	400mW ~4W
2×AA 电池	4000	0.5 年（10% ~20% 使用时间）	1 ~10mW
钮扣电池	40	5 年（一直用的情况下）	1μW

对于需要在电池上运行额外一段时间的SoC设计来讲，整个系统的功率必须保持很低（大约为毫瓦级）。因此，功率管理的实现也就从系统架构和操作系统级延伸到了逻辑门级。

另外一个功率上的约束峰值功率也是设计者不能忽略的一个问题。在任何设计当中，在特定电压下电源只能提供相应的电流；超过这个值（即使是一个瞬变）都会导致逻辑错误或者更糟糕的状况（毁掉电源）。

2.6 在处理器设计中面积-时间-功耗的权衡

对于以下两种常见的处理器来讲，设计权衡是相当不同的：

1. 工作站处理器。这类设计是以高频和交流电源为设计导向的（不包括笔记本）。因此它们没有面积限制，缓存占据了芯片的大部分面积，设计是相对非常精心复杂的（超标量多线程）。

2. SoC上的嵌入式处理器。这类处理器通常控制结构比较简单，但是在专用处理器中可能会比较复杂（如DSP）。面积是一个重要因素，设计时间和功耗同样也是。

2.6.1 工作站处理器

为了得到通用目标的性能，设计者假定电源能量是足够的。最基本的权衡是在时钟的高频率和导致的高功耗之间的权衡。早在20世纪90年代初期，利用了双极工艺的射极耦合逻辑（Emitter Coupled Logic，ECL）在高性能应用领域（主体架构和超级计算机）占据了主导地位。其功率密度为80W/cm^2，模块的封装需要液体的散热形式。日本日立公司的M-880是这个时期的一个例子（见图2.14），一个约10cm×10cm的模块功率为800W。这个模块包括40个分割块，用冷水密封在氦气当中，这些冷水是从模块上方的一个冷却管压入的。由于CMOS是接近双极性的，所以这样一个冷却系统的特殊成本就不复存在了，然后晶体管时代结束了（见图2.15）。现在，CMOS已经达到了相同的功率密度，如果芯片频率继续增加，那么就不得不再次考虑使用类似的冷却技术了。事实上，在2003年以后，有

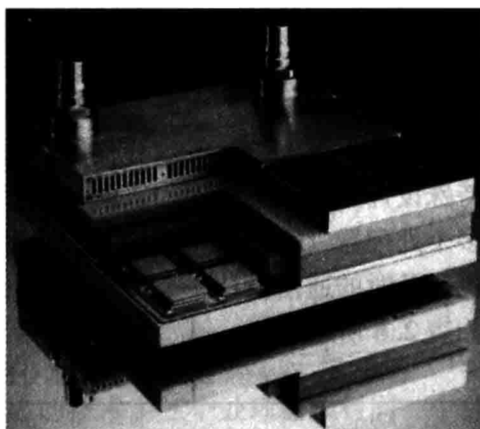


图2.14 日本日立公司M-880处理器模块（M-880大约是在1991年问世的。该模块大小为10.6cm×10.6cm，功率为800W并带有水冷系统）

用的芯片频率稳定在了 3.5GHz 左右。

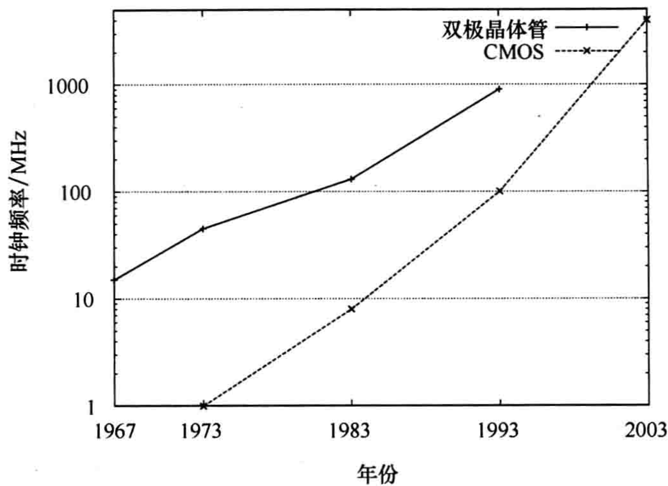


图 2.15　不同时间双极晶体管和 CMOS 处理器频率

（一般来讲，CMOS 频率的增长已经因为功率而受到约束，在大约 2003 年停在了 3.5GHz）

2.6.2　嵌入式处理器

片上芯片的实现有很多的优势。其需求是众所周知的，所以存储器大小和实时延迟约束是可以预测的。处理器可以为了实现一个特定功能而特色化。通常，这样做可以降低时钟频率（功耗），同时可以通过直接开发体系结构内的并发性重新得到相应性能（如 DSP 应用中使用简单 VLIW）。跟处理器芯片相比，可利用的设计时间/工作和芯片内部功能块间的通信是 SoC 的劣势。对任何一个特定系统，SoC 的市场都相对较小。因此，人们很难承担在处理器芯片上广泛的定制优化，所以通常都会用成品的处理器核。因为在设计的时候就知道程序和数据所需的存储空间大小，特定存储结构可以被包含在芯片上。这可以是 SRAM 或者特殊设计的 DRAM（因为普通 DRAM 的工艺与之不相容）。有了多个存储单元、多个处理器（有些是专用的有些是通用的）和特殊的控制器，问题就只剩下为保证及时的通信设计一个稳健的总线层了。典型的处理器芯片与 SoC 芯片的比较如表 2.6 所示。

表 2.6　典型的处理器芯片与 SoC 芯片的比较

	单芯片处理器	SoC
存储所用面积	80% 的缓存	50% 的 ROM/RAM
时钟频率	3.5GHz	0.5GHz
功率	≥50W	≤10W
内存	≥1GB DRAM	大部分在

2.7 可靠性

第四个重要的设计因素是可靠性[199]，也叫做可依赖性或容错性。影响可靠性的因素比成本和功耗对处理器或 SoC 芯片的影响因素要多很多。

可靠性跟芯片面积、时钟频率和功耗有关。芯片面积增加了电路的数量和发生错误的可能性，但它也允许使用错误探测和纠错技术。高时钟频率增加了电气噪声和噪声密度。越小的电路速度越快，也就越容易受辐射的影响。

不是所有的失效和错误都会导致故障，确实，也不是所有的故障都会导致不正确的程序执行。故障如果被探测到，就可以用纠错码（Error Correcting Code，ECC）标识，指令重新执行或者重新配置。

首先说明以下几个定义：

1. 失效（failure）是与设计规格的偏差。
2. 错误（error）是导致不正确的信号值的失效。
3. 故障（fault）是一种表现为不正确逻辑的错误。
4. 物理故障（physical fault）是由环境引起的，如老化、辐射、温度和温度周期变化等。物理错误的可能性是随时间增加的。
5. 设计故障（design fault）是由设计实现与设计规格不一致而导致的失效。通常，设计错误在设计的早期产生，随时间慢慢降低或者被消除。

2.7.1 解决物理错误

从系统的角度，需要生成的处理器和其附加配置是必须一直很稳健的。假设一个错误发生的概率为 $P(t)$，T 为错误的平均时间（Mean Time Between Fault，MTBF）。所以，如果 λ 是错误率，则

$$\lambda = \frac{1}{T}$$

那么假设错误发生在特定单位的时间轴上，时间轴被平均时间 T 分割开来。利用与建立的成品率泊松方程相同的推理，可以得到错误泊松方程：

$$P(t) = e^{\frac{-t}{T}} = e^{-t\lambda}$$

冗余是一种能够很明显提高可靠性（降低 $P(t)$）的方法。三模冗余（Triple Modular Redundancy，TMR）是一种很著名的技术。三个处理器执行同一个计算并进行结果比较。一个裁决器选出输出结果，这个结果必须是由至少两个处理器同意的。TMR 能够起作用，但仅是一定程度上起作用。除了很显然的裁决器的可靠性问题外，绝对数量的硬件也有问题。很明显，虽然由 t 能够得到 T，但是在 TMR 系统中错误的数量是可以比单纯型系统中多的（见图 2.16）。当然，

当 t 满足如下公式时，TMR 中错误的概率是超过单纯型系统的：

$$t = T \times \log_e 2$$

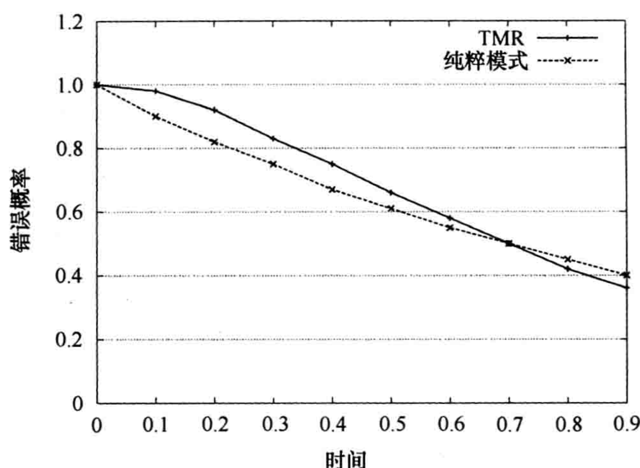


图 2.16 TMR 与单纯型系统的可靠性比较

大部分容错设计包括了如下更简单的硬件：

错误检测。对于可靠系统配置，奇偶码、残余码和其他编码的使用是非常有必要的。

指令（操作）重试。一旦检测到错误，操作就重新执行，以解决瞬变错误。

错误校正。既然系统的大部分是具有存储功能的，利用 ECC 可以有效地克服存储错误。

重新配置。一旦错误被检测到，很可能系统的一部分就需要重新配置，这样的话发生错误的子系统就与后面的计算隔离开来。

需要注意的是，带错误检测的高效可靠的系统配置是有限的。作为最低限度，大部分系统都需要在必要的系统部件的所有部分安装错误检测，并且应该有选择性地使用 ECC 和其他技术来提高可靠性。

IBM 大型机 S/390（见图 2.17）系统就是一个以可靠性为导向的实例。模型包括了 12 个处理器模块：5 对双工组态（5 ×2）运行了 5 个独立的任务，2 个处理器用来监控和备用。在双工状态下，成对处理器共享通用的缓存和存储系统。成对处理器运行相同的任务并且对结果进行比较。处理器在所有可能的地方用了错误检测。缓存和存储系统利用的 ECC，通常是单错校正、双错检测（Single Error Correction Double Error Detection，SECDED）的。

近期的研究将可靠性落脚于多处理器 SoC 工艺上。例如，通过宇宙射线的单一事件干扰来提高可靠性，电压调整和应用任务映射等技术也将会用到[199]。

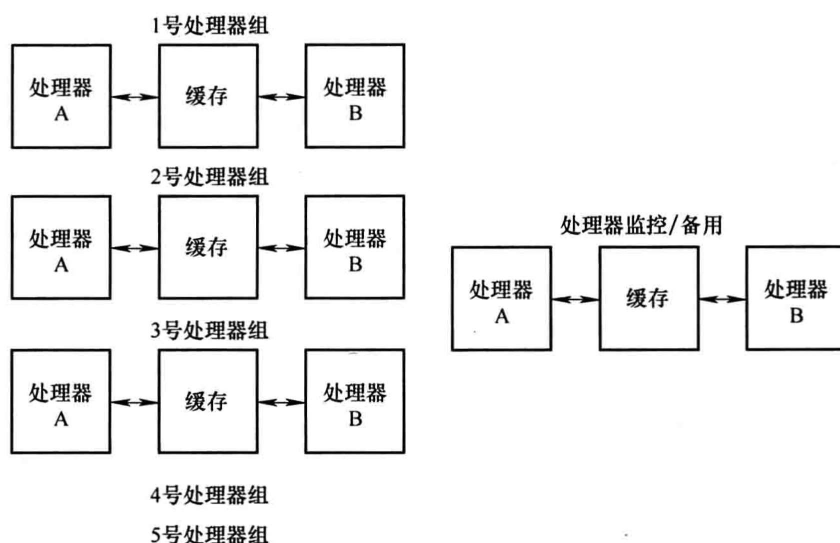


图 2.17　一种利用错误探测的双工容错方法

2.7.2　错误检测和纠正

最简单的错误校验方式是奇偶校验。将 1 位数字（校验位）加到每个存储的字或传输的数据中，保证这个字中 1 的个数之和为偶数（或者奇数，根据方便提前定义）。如果字中的任何一位发生了一个错误，那么字中 1 的个数之和对 2 取模就会跟奇偶假设不一致，那么存储的这个字就被认为是有误的。

知道被检索的这个词中有错误是很有价值的。通常，简单的重复接收一次就可以得到正确的内容。然而，通常在特定存储器单元中的这个字已经丢失，再多次的重复接收也无法得到数据的正确值。因为这种错误往往在大的系统中发生，所以大部分系统都包含使用 ECC 方法的自动纠正单错的硬件。

这种形式的最简单编码是由几何块代码构成的。要被检测的比特报文被排在近似方形的网格中，用校验位对每一行和每一列进行增广。如果一行和一列表示了报文在接收端解码时发生了错误，那么交叉点就是出错位，简单的取反就可以得到正确结果。如果仅有一行或者仅有一列或者多行或多列说明校验失败，那么多比特错误可以被检测到并且会加入一个未纠正状态位。

对于一个 64 位报文，需要加 17 个校验位：八个行校验位八个列校验位和一个额外的校验行列校验位的校验位。（见图 2.18）。

将报文的每一位想成是用来构成超立方体会更有效一些，每个报文在这个超立方体组合中形成一个特定的点。如果这个超立方体可以扩展，这样合法的数据点可以被相关的引起单位错误的不合法点所围绕，那么解码器机可以识别出不合

法点属于哪个合法点并且将会存储报文原始正确的扩展形式。在两个合法数据点之间再加一个不合法点需要再延伸一步（见图 2.19）。与合法码字不同的最小位数称为码距。这第三个点就表明发生了两个错误。因此，这两个合法数字编码点很可能是相等的，所以报文是可以检测错误但是不能纠正的。对于一个 64 位报文，一位的错误校正，每 2^{64} 个组合对中一定会围绕或者包含这 64 个组成位的一种失效。因此，需要总共 2^{64+6} 个代码组合来识别每个合法状态位相关的不合法状态位，或者总共需要 2^{64+6+1} 个数据状态。可以用另外一种表达方式：

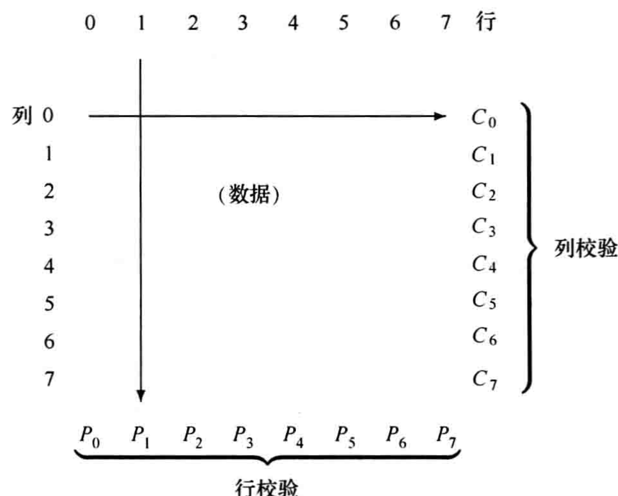


图 2.18 二维 ECC

$$2^k \geqslant m + k + 1$$

式中，m 为报文的位数；k 为支持单错纠正所必须加入的正确的位数。

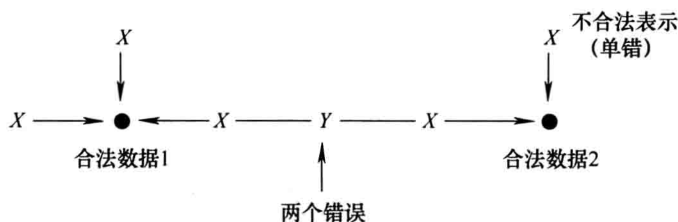


图 2.19 ECC 的码距

汉明码代表了基于超立方体的 ECC 的一种实现。与之前在块编码中一样，一对等价失效可以定位一个错误位的位置。在汉明码中，k 个正确位确定了一个错误位的位置。报文必须排布的能够为代码提供正交基（就像块编码中的行和列一样）。而且，正确位必须包含在这个基中。例 2.1 给出了一个 16 位报文的正交基和 5 个已经确定的正确位。加入另外一个第 6 位后，就可以计算校验整个 $m+k+1$ 位报文。如果从这 k 个正确位得知有 1 位可纠正位，并且得知这新的 d 位中没有校验错误，那么就可以判断有两位错误，并且任何形式的纠错都可能会造成错误，所以不应该尝试纠错。这种编码通常被称作 SECDED。

例 2.1 一个汉明码实例。

假设有一个 16 位报文，$m=16$。

$2^k \geqslant 16+k+1$；所以，$k=5$。

那么，报文就有（16+5）位=21位。这5个纠错位是根据奇偶校验如下定义的，基2超立方体：

k_5　16~21位

k_4　8~15位

k_3　4~7，12~15和20、21位

k_2　2、3，6、7，10、11，14、15和18、19位

k_1　1，3，5，7，9，…，19，21位

换句话说，21位形成的报文f_1~f_{21}位包含了原始信息位m_1~m_{16}和纠错位k_1~k_5。每个纠错位都在它所检验的组内。

假设报文包含了f_1~f_{21}和m_1~m_{16}=0101010101010101。为了解码简单，把纠错位仅放在覆盖制定的纠错码位置（如只有k_5覆盖第16位）：

$$k_1 = f_1$$
$$k_2 = f_2$$
$$k_3 = f_4$$
$$k_4 = f_8$$
$$k_5 = f_{16}$$

这样就有（m_1在f_3，m_2在f_5等）

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}	f_{21}
k_1	k_2	0	k_3	1	0	1	k_4	0	1	0	1	0	1	0	k_5	1	0	1	0	1

所以，根据奇偶校验，

$$k_1 = 1$$
$$k_2 = 1$$
$$k_3 = 1$$
$$k_4 = 0$$
$$k_5 = 1$$

假设这个报文传递但是接收到的$f_8 = 0$（本来它应该是$f_8 = k_4 = 1$）。当接收器对着5个校验组进行重新计算时，只有一组包含了f_8。

对组内进行重新奇偶校验得到：

$k'_5 = 0$（在16~21位中没有错误）

$k'_4 = 1$

$k'_3 = 0$

$k'_2 = 0$

$k'_1 = 0$

失效形态01000错误位（第8位）的二进制表示，必须改变这一位来纠正报文。

2.7.3 解决制造缺陷问题

传统的解决制造缺陷问题的方法是测试。随着晶体管密度的增加和整个芯片晶体管数量的相应增加，测试的问题发展得更快。可测试的组合对随着晶体管的数目呈指数性增长。如果没有测试方面的突破，据估计，芯片测试的费用将会超过其余的制造费用。

如果有可能的话，先验性的确定设计的完整性是一件很难的事情。依据设计验证的等级，不同的自动设计工具是很有用的。在某些情况下，当设计完成，设计的逻辑模型也已经被验证。设计验证包括比较设计的逻辑输出和设计的预期逻辑需求是否一致。在像存储（如缓存）甚至是浮点运算器区域内，合理的验证实现是可能的。更一般的验证是正在进行研究的课题。

当然，硬件设计者通过可测试性设计过程也能做出测试和验证方面的努力[104]。可用的错误侦测硬件就是很显然有助于测试的。一种可以得到内部存储单元信息的测试技术叫做扫描（scan）。一个扫描链最简单的形式包括对每一个存储单元有一个分开的入口点和一个出口点。每个点都通过多项选择器连接到一些总线上，这样就可以独立地通过下载或存储来测试系统。在扫描技术中，允许将预定义的数据配置送入存储单元，然后将输出特定的配置跟已知的正确输出配置进行比较。扫描技术开始于20世纪60年代，是作为大型机技术的一部分而发展起来的。它们后来大部分都被遗弃了，直到在高密度块出现的时候才被再次发掘。

扫描链需要大量的测试配置来覆盖这么大的设计。因此，即使是扫描技术，对设计验证的潜力也是有限的。通过压缩模式所需要的数量和结合各种内置自检功能，可以扩展更新的扫描技术。

2.7.4 存储和功能擦除

擦除技术（scrubbing）是一种在空闲或不可用（如启动）的时候对一个单元进行演练性测试的技术。它经常被用于存储器。当存储器空闲的时候，存储单元就被重复地进行读写操作。这种检测可以探测到存储器坏掉的部分，然后将这部分声明为不可用，进程再加载的时候就会避开。

原则上，相同的技术也可以被用在功能部件上（如浮点部件）。很明显，如果可能有个重新配置单元可以让系统操作继续的话，那么将会更有效（在降低了性能的情况下）。

2.8 可配置性

这部分涉及两个主题，包括可配置性，主要关注设计的可再配置性。第一，

列举几条可再配置性设计的动机，包括一个简单的例子来说明这个基本的观点。第二，估计了一下基于本章前部分介绍过的 rbe 模型并发性可配置器件所需要的面积成本。

2.8.1　为什么要可配置性设计

本书第 1 章介绍了采取可配置设计的动机，但主要是从基于对管理高性能的 IP 核的复杂度和避免跟制造相关的风险和延迟的角度出发的。这里对使用可配置性器件，如 FPGA，再说明三点原因。这三点原因基于本书前面介绍的时间、面积和可靠性方面的内容。

时间。因为 FPGA 尤其是细粒度的 FPGA 包含有大量的寄存器，所以它们支持高流水线设计。另外一个考虑就是并行性。一个基于 FPGA 的低时钟频率处理器通过定制电路的并行执行，能够有相似甚至更高的性能，而不是以高速时钟频率来运行一个顺序处理器。相反的，微处理器的指令集和流水结构并不一定适合一个给定的应用程序。后面将会通过一个简单例子来说明这一点。

面积。FPGA 的可编程性确实会带来面积开销，但是 FPGA 比 ASIC 更加简化了对先进制造工艺技术的应用。所以，FPGA 比其他形式的电路更能够探索先进的工艺。此外，一个小的 FPGA 可以通过时间分割选择器和时间的配置来支持一个很大的设计，配置可以允许选择执行时间和所需要的资源数量。下面也将估计一下一些基于本章前面介绍的 rbe 模型的 FPGA 设计的大小。

可靠性。FPGA 的规整性和同质性使得更多的单元和内部连接可以加到其结构当中。通过这种冗余的结构，产生了很多避免制造或者运行错误的不同方法。此外，在不同的半导体制造工艺中，FPGA 的可配置性已经成为了一种提高电路成品率和时钟的方法[212]。

为了说明用 FPGA 加速应用需求的可能，这里介绍一个比较微处理器和 FPGA 运行 HDTV 程序的简单例子。HDTV 规格为 1920 × 1080 个像素，或者大约为两百万像素。在 30Hz 时，相应的需要每分钟六千万像素。一个特定的应用包括了 100 个操作，所以进程每秒钟大约需要 60 亿次操作。

思考一下，一个 3GHz 的微处理器平均需要 5 个周期来完成一个操作。那么，它可以支持每周期 0.2 个操作，总体来说，每秒钟只能执行 6 亿个操作，是只所需要的进程频率的 1/10。

相反的，一个 100MHz 的 FPGA 设计每周期可以并行完成 60 个操作。这个设计能够达到程序每秒钟 60 亿操作的要求，是 3GHz 微处理器的十倍之多，虽然，它的时钟频率仅是微处理器的 1/30。这种设计可以以不同的方式利用可配置性，如针对特定的执行数据利用实例制定优化来提高面积、速度和功耗，或者重新配置使设计适应执行时间的条件限制等。对可配置性更深入的讨论将会在本

书第6章。

2.8.2 可配置器件的面积估计

为了估计可配置器件的面积，利用前面讨论过的rbe作为基本方法。回顾一下，例如，在实际设计中，六管寄存器单元需要$2700\lambda^2$。

在Xilinx FPGA的一个“slice”中，配置、布线和逻辑大概需要7000个晶体管，在一个Altera器件的逻辑单元（LE）中大概有12000个晶体管。根据经验，每个rbe包含大约10个逻辑晶体管，所以每个slice包含700个rbe。一个大的Virtex XC2V6000器件包含33792个slice或者说2365万个rbe或者16400A。

在这种工艺下，一个8×8的乘法器将会有35个slice或24500个rbe或者17A。相反的，假设包含一个全加器和一个与门的一位乘法器单元在VLSI技术中有大概60个晶体管，相同的乘法器将会有64×60个晶体管=3840个晶体管，即384rbe，这只是可配置版本的大概1/60。

考虑到在设计中乘法器经常会用到，现在很多FPGA有专门的资源来支持乘法器。这种方法将可配置资源空出来来实现其他功能而不是乘法器，代价就是使得器件不再那么规则，并且当设计中用不到它们的时候会产生面积浪费。

2.9 总结

在处理器设计中，周期时间是最重要的。它很大程度上取决于工艺但是主要受次要因素的影响，如时钟问题和流水分割等。

一旦周期时间被确定，设计者的下一个挑战就是通过使用最小的芯片面积且使面积得到最好的性能，来优化性价比。一个独立于工艺的衡量面积的单位叫做rbe，它为在一系列的重要结构因素中权衡存储层次提供了基础。

虽然高效的利用芯片面积是很重要的，芯片的功耗也同等重要（有时候甚至更重要）。权衡性能-功耗比更注重将所需时钟频率最小化，因为功耗跟频率的二次方呈函数关系。因为低功耗可以应用于很多应用环境，尤其是那些无线和基于遥感的环境，所以认真地对器件功耗进行优化决定了设计是否成功，尤其是在SoC设计中。

可靠性通常是个假定的要求，但是更小的工艺尺寸使得设计对辐射和类似的冒险更加敏感。

根据应用，设计者必须预执行冒险并结合特征来保存计算的完整性。

SoC设计的最大难题就是在预算约束范围之内如何充分利用工艺优势。可配置性当然是新兴的一种很有用的方法，尤其是所选择的FPGA技术。

2.10　习题

1. 用四路流水来实现一个函数，每一路的延时如下（$b=0.2$）：

段　序　号	最大延时*/ns	段　序　号	最大延时*/ns
1	1.7	3	1.9
2	1.5	4	1.4

* 执行时钟开销为 0.2ns。

（a）不加入乘法周期的情况下，可以为最高性能提供多大的周期时间？

（b）执行这个函数的总时间是多少（通过所有的步骤）？

（c）如果每级流水可以分解成次级流水，那么能够为最高性能提供多大的周期时间？

2. 如果在每个时钟脉冲到来时有 0.1ns 的时钟歪斜（不确定性为 ±0.1ns），回答习题 1 中的问题。

3. 通过允许流水中断延时 $S-a$ 个周期（而不是 $S-1$ 个周期）来一般化 S_{opt} 的方程，此时 $S>a\geqslant1$。写出 S_{opt} 新的表达式。

4. 一个流水线有如下功能单元和延时（不包含时钟开销）：

功 能 单 元	延时/ns	功 能 单 元	延时/ns
A	0.6	D	0.7
B	0.8	E	0.9
C	0.3	F	0.5

功能单元 B、D 和 E 可以细分成两个相等的步骤。如果预期的并发流水线的延时为 $b=0.25$ns，时钟开销为 0.1ns，那么：

（a）最优化的流水分割数量为多少？（下舍入为整数值）

（b）这样周期时间为多少？

（c）利用这个周期时间计算流水线性能。

5. 一个处理器芯片（1.4cm × 1.4cm）需要生产 5 年。在这段时间内，预计缺陷密度将会线性地从 0.5 个缺陷/cm^2 降为 0.1 个缺陷/cm^2。20cm 的晶圆成本将会线性地从 5000 美元降到 3000 美元，30cm 的晶圆成本将从 10000 美元降为 6000 美元。假设每年生产好的器件数是常数。那么应该选择哪个产品工艺呀？

6. 深度优化减小单元尺寸和数据存储开销的 DRAM 芯片设计是一门特殊的艺术。一个尺寸为 135λ^2 的单元，计算 DRAM 芯片的容量。工艺参数为，产率

为 80%，$\rho_D=0.3$ 个缺陷/cm²，特征尺寸为 0.1μm，开销中有 10% 的驱动和放大开销，端口、驱动和保护环等的开销为 20%，没有总线和触发器。

存储器的尺寸必须是偶数幂 2，计算容量并重新计算芯片的尺寸，算出实际总面积（除去浪费的空间）和相应的成品率。

7. 利用习题 6 中的假设，计算 $512^M \times 1^b$ 的芯片的成本。假设一个直径为 30cm 的晶圆的价格为 15000 美元。

8. 假设一个 2.3cm² 的芯片可以以 5000 美元的成本在一个 20cm 晶圆上完成，也可以以 8000 美元的成本在 30cm 的晶圆上完成。在缺陷率分别为 0.2 个缺陷/cm² 和 0.5 个缺陷/cm² 的情况下比较单个芯片的有效成本。

9. 根据成品率方程的推导过程，证明：

$$P(t)=e^{\frac{-t}{T}}$$

10. 证明：3 模块系统的 2 个模块失效预执行时间 t 为

$$t=T\times\log_e 2$$

提示：有 3 个模块，任意 2 个或 3 个模块失败，则系统失败。

11. 为 32 位的报文设计汉明码。将检测位放在最终报文中。

12. 假如想为 64 位报文设计一个双错矫正汉明码。那么需要多少个检测位？请解释。

第3章 处 理 器

3.1 引言

处理器有多种类型和不同的用途，虽然很多人都将目光聚焦在服务器和工作站上运行的高性能处理器，但从数量上讲，高性能处理器只占每年生产处理器的一小部分。图 3.1 所示为世界范围处理器年产量概况（不根据价格）。

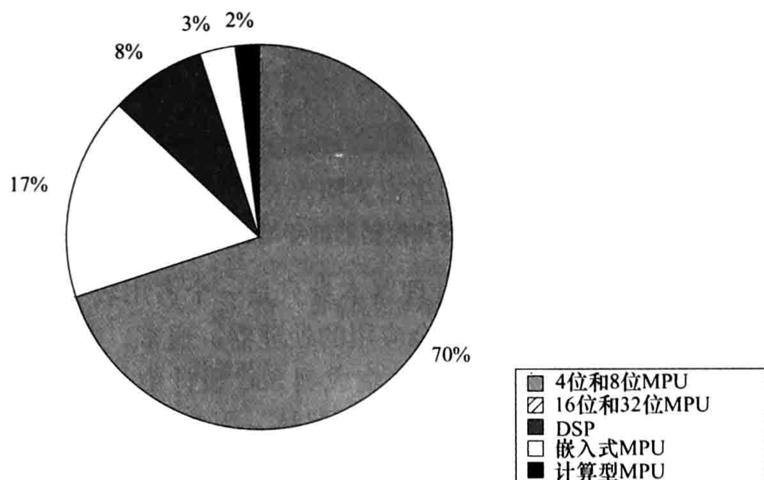


图 3.1 世界范围处理器年产量概况[227]

本章和处理器细节

本章包含处理器设计上的问题，特别是面向高性能应用的高级处理器。已经确定所选处理器的读者以选择跳过一些细节，如分支预测和超标量体系结构控制这些部分，这些章节已用星号(*)注明。

这些细节对于选择处理器的读者来说也是重要的，主要由于以下两个原因：

1. 年复一年，SoC 处理器和系统都在向越来越复杂的方向发展，SoC 的设计者需要面对这些越来越复杂的处理器。

2. 处理器性能评估工具（如 SimpleScalar[26]）提供指定问题的选项，如分支预测和相关参数。

对面向应用指令处理器（见本书 1.3 节）感兴趣的读者，可以在本书 6.3 节、6.4 节、6.8 节找到相应的材料。

很明显，控制器、嵌入式控制器、数字信号处理器等是占统治地位的处理器，大部分的精力都放在设计这些处理器上。市场的增长规律的数据也能同样说明 SoC 和大型微处理器的需求在以接近 3 倍的微处理器单元（Microprocessor Unit，MPU）的速度增长（见图 3.2）。

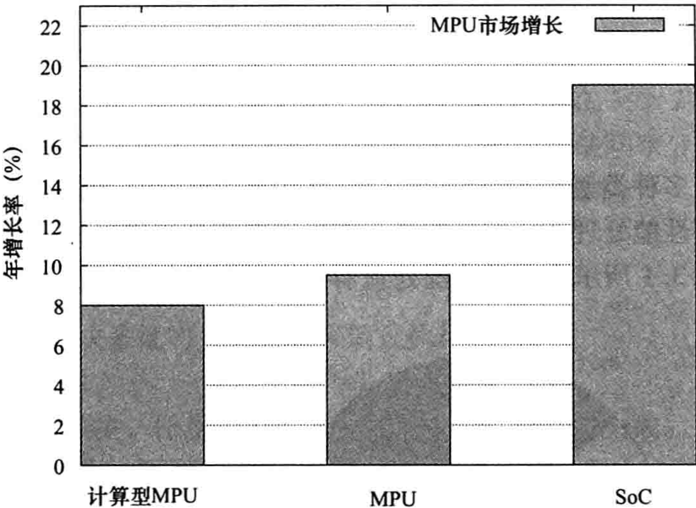


图 3.2 微处理器和控制器的年增长情况[227]

特别在 SoC 类型的应用方面，处理器本身只是一个占用较少面积的小组件，SoC 设计经常要运用许多不同类型适合应用的处理器。通常，非关键处理器以设计文件（如 IP）的形式获取（购买）并集成到 SoC 设计中。因此，一个专用的 SoC 设计可以整合其他人士设计的通用处理器核。表 3.1 给出了利用 IP 优化设计以获取更优的面积-时间性能，SoC 设计中选择不同级别 IP 有各自的相对优势。

表 3.1 利用 IP 优化设计以获取更优的面积-时间性能

设计类型	设计级别	相关预期面积×时间
定制 IP 硬核	纯物理级	1.0
综合后的 IP 固核	物理级	3.0～10.0
IP 软核	RTL 或 ASIC 级	10.0～100.0

3.2 SoC 处理器的选择

3.2.1 概述

对于许多 SoC 设计的情况，选择处理器是最重要的工作，而且在某种意义上是最受限的工作。处理器必须运行一个特定系统软件，所以为了满足这一功能，至

少需要选择一个处理器，通常是通用处理器（General Purpose Processor，GPP）。在计算限制的应用中，最基本的初始设计推力是确保系统包含一个配置和设置参数后能满足需求的处理器。在某些时候，可能会结合这些处理器，但这通常是一种优化的考虑，应该在后续工作中进行。在决定处理器性能和系统性能时，将内存和互联组件当做一种简单的延迟元素，这些被视为理想化组件，因为它们的行为被简化，但这种理想化需要保证结果特性是可以实现的。理想化的组件的性能应该保守估计。

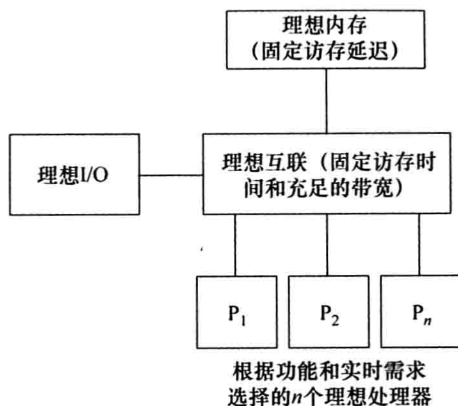


图 3.3　SoC 上的处理器模型

图 3.3 所示为 SoC 上的处理器模型，是初始化设计流程中的处理器模型。图 3.4 给出了选择处理器核的流程。选择计算限制型应用处理器时的流程有所不同，因为可能需要考虑实时性需求，它必须由一个所选处理器实现，在初始化 SoC 设计阶段的早期就应该成为主要的考虑要素。处理器的选择和配置应该产生一个看似完全满足规格说明上列出的性能和功能需求的初始化 SoC 设计。

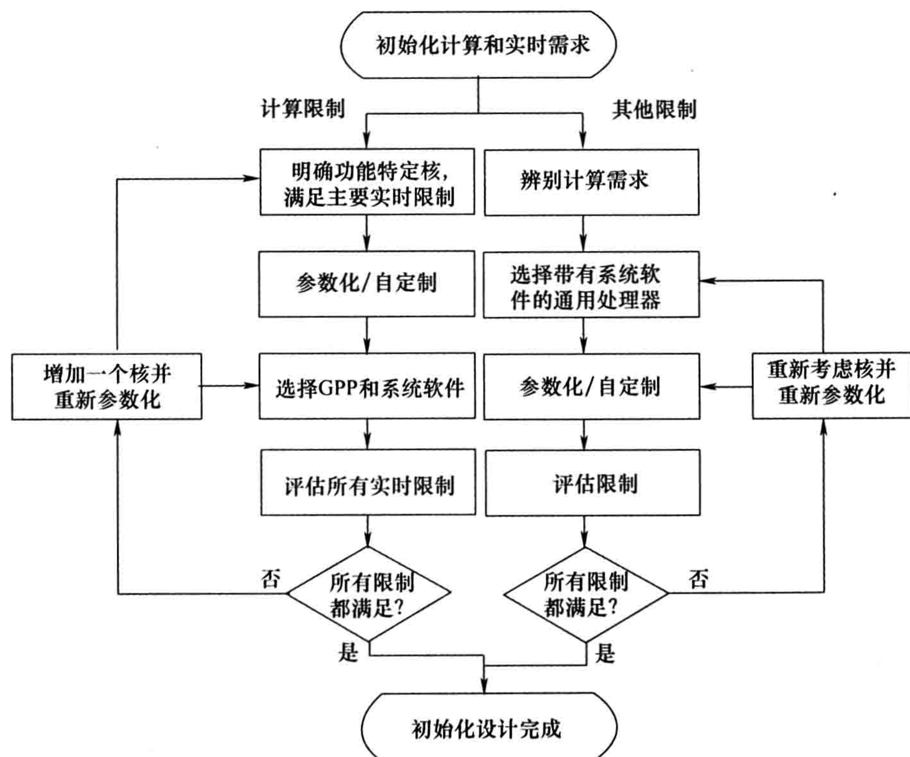


图 3.4　选择处理器核的流程

3.2.2 实例：软处理器

“软核”指的是一个比特流形式的指令处理器设计。它可以用于编程 FPGA 设备，尽管这需要大量的面积、能耗、时间的开销。但运用这样的设计有以下 4 个主要的原因：

1. 降低系统级综合的成本。
2. 在仅有一点不同的多种设计中进行设计重用。
3. 产生一个微处理器/外围设备组合的精确匹配。
4. 在未来提供保护以防止微控制器的不间断更新。

主要的指令处理器软核包括以下几个：

Nios II[12]：由美国 Altera 公司开发，用于它们的 FPGA 系列和 ASIC。

MicroBlaze[258]：由美国 Xilinx 公司开发，用于它们的 FPGA 和 ASIC 系列上。

OpenRISC[189]：一个免费开源的软核处理器。

Leon[105]：另一个免费开源的软核处理器，实现了一个完全的 SPARC v8 指令集结构。另外它还包含一个可选的高速浮点部件 GRFPU。GRFPU 可以下载但是并不开源，只能用于评估和研究。

OpenSparc[228]：这是一个 SPARC T1 核，它在 FPGA 上支持单线程和四线程。

这些指令处理器都有很多显著特征，但实际上，它们都支持一个 32 位 RISC 体系结构（OpenSPARC 除外，它是 64 位的），都包含单发射五级流水级、可配置的数据/指令缓存，同时支持 Gnu，编译套装（Gnu Compiler Collection，GCC）编译工具链。它们还有一种总线结构，适合增加额外的处理单元当做主单元或从单元来加速运算，还有一些处理器更进一步允许增加自定指令或协处理器。

表 3.2 给出了一些软处理器的特征，针对不同 SoC 特征进行了简单的比较。值得注意的是，MIPS 的测量数值是从市场材料获得的，可能会根据特定处理器的具体配置出现较大程度的波动。

表 3.2 软处理器的一些特征

	Nios II（快）[13]	MicroBlaze[259]	OpenSPARC[187]	Leon4[105]
开源	否	否	是	是
硬件 FPU	是	是	否	是
总线标准	Avalon	CoreConnect	WISHBONE	AMBA
定点除法部件	是	是	否	是
自定协处理器（指令）	是	是	是	是
FPGA 上最高频率/MHz	290	200	47	125

（续）

	Nios II（快）[13]	MicroBlaze[259]	OpenSPARC[187]	Leon4[105]
FPGA 上最大 MIPS	340	280	47	210
资源	1800 个逻辑单元	1650 个 slice	2900 个 slice	4000 个 slice
估计面积	1500A	800A	1400A	1900A

简单比较一下在本节前半部分评估的一个 32 位处理器，在没有浮点部件的情况下面积为 60A，只是表 3.2 所示的软处理器的 1/30 ~ 1/15。

3.2.3 实例：处理器核选择

下面用两个例子来阐明图 3.4 所示的步骤。

例 1 处理器选择，通用核的路径。

考虑图 3.4 所示的“其他限制”的路径并关注一些关系的权衡，对于这种简单的分析方法，不需要去理会处理器的细节，只需要假设处理器遵循本书第 2 章所述的 A-T^2 规律。假设一个初始化设计拥有单位 1 的性能，运用了 100Krbe 的面积，想要更高的速度和更多的功能，那么如果我们将性能加倍（将处理器的 T 减小一半），这将使面积增加到 400Krbe 并且功耗增加 8 倍，每个 rbe 像以前一样对应两倍的功耗，所有的性能由内存系统调控。将性能（指令执行率）加倍的同时会使单位时间缓存失效次数加倍。这一现象对已实现的系统的性能影响显著，它依赖平均缓存失效时间。更多的介绍可以参见本书第 4 章。

假设缓存失效现象显著降低实现系统的性能，为了恢复性能，需要增加缓存的容量。本书第 4 章引用了一个一般规律，降低一半的缓存失效率需要增加一倍的缓存容量。如果初始化缓存容量也是 100Krbe，那么新的设计需要 600Krbe，并且它的功耗可能对应初始化设计的 10 倍左右。

这样是否值得呢？如果面积充足且功耗并不是一个显著的约束，那么可能这是值得的。更快的处理器缓存组合可以提供重要的功能，如额外的安全检查或输入/输出（I/O）能力。在这一点上，系统的设计者要参考设计规格说明。

例 2 处理器选择， 计算核路径。

如图 3.4 所示，目前只考虑了计算限制型路径的权衡比，假设应用通常情况下是可并行的，并且已有一些不同的设计方式。一种是 10 级流水的矢量处理器，另一种是多个简单处理器。应用在矢量处理器（面积为 300Krbe）上的性能为 1，在单个简单处理器（面积为 100rbe）上的性能为 0.5。为了满足实时计算的需求，需要将其性能增加到 1.5。

这时必须评估实现性能目标的各种方法：方法一，增加流水级级数并将矢量

流水线数量加倍，这样满足了性能的目标，面积达到了600Krbe并且功耗翻倍，而时钟率频率保持不变；方法二，运用一个互联的简单处理器“阵列”，这种多处理器阵列受限于内存和互联竞争（本书第5章有更多这些效应的详述）。为了实现性能目标，需要至少4个处理器：3个用于基本目标，1个用于处理额外开销。这时的面积为400Krbe，加上互联面积和额外的内存共享电路，这可能还要增加另外的200Krbe。所以仍然是有面积或功耗方面一致的两种方法的。

那么怎样从两者中挑选呢？通常可供选择的数量比2更大。这就取决于次要的设计目标，下面列出需要考虑的事项：

1. 应用可以简单地被划分为同时支持两种方法吗？
2. 两种方法有哪些软件支持（编译器、操作系统等）？
3. 可以运用多处理器方法得到额外的容错吗？
4. 多处理器方法可以和其他计算路径结合吗？
5. 两种设计改进方法中的一种是否有显著的设计实现成本？

显而易见，系统设计者有很多问题需要回答，工具和分析仅消除了不符合要求的方法，在这之后，真正的系统分析开始了。

本章下面的部分涉及对处理器的理解，特别是微处理器的级别和它怎样影响性能。这对于评估性能和应用模拟工具是十分重要的。

达到性能指标的方法

3.2节给出的重要应用的实例是一个非常简化的模型。那么应该怎样设计来符合$A\text{-}T^2$设计规律或者实现更好的性能呢？其中的秘诀在于理解设计的可行性，并自如地从设计备选方案中选择。为了实现这一点，必须理解现代处理器的复杂度和它所带来的衍生物。本章给出了大量的备选方案，但仅列出了最重要的，还许多其他技术可能在特定环境下对设计者有效。理解的作用是无法替代的。

3.3 处理器体系结构中的基本概念

处理器体系结构由处理器指令集组成。虽然指令集暗指许多实现（微体系结构）细节，具体的实现相比指令集需要的还要更多。它是物理设备面积-时间-功耗限制权衡的综合，用于优化特定的用户需求。

3.3.1 指令集

对于大多数处理器来说，指令集基于一组用于存放操作数和地址的寄存器，这些寄存器组的大小为8～64字甚至更多，每一个字由32～64位组成。另外，

还经常会有一组额外的浮点寄存器（32～128位）。一个典型的指令集会制定一个程序状态字，它由多种类型的控制状态信息组成，包括由指令设置的条件码（Condition Code，CC）。普通指令集可以被分为两种基本类型：加载/存储（L/S）体系结构和寄存器/内存（R/M）体系结构。

L/S体系结构的指令集包括RISC微处理器。在执行前参数必须已经存在寄存器中。ALU指令的源操作数和目标都指定在寄存器中。L/S体系结构的优点在于执行的规则性和指令译码的简单性，一个简单时序的简单指令集易于实现。

R/M体系结构包含的指令的操作数位于寄存器或者其中一个内存中。在R/M体系结构中，一个ADD指令可以将一个寄存器的值和内存中的值相加，结果存在一个寄存器中。R/M指令集是由IBM大型机和Intel x86（现在成为Intel IA32）演变而来。

指令集的权衡是针对面积-时间的折中。R/M方法提供了更精确的程序表现，较L/S方法相比用到了更少的不同长度的指令，程序在内存中占据更少的空间，指令缓存也可以因此做得更小。多种指令长度对于译码而言更加困难，多种指令的译码需要预测每一条指令的起点，R/M处理器需要更多的电路（和面积）用于取指和译码。通常，Intel x86在高时钟频率下的成功实现表明没有一种方法的优点可以完全战胜另一种方法。

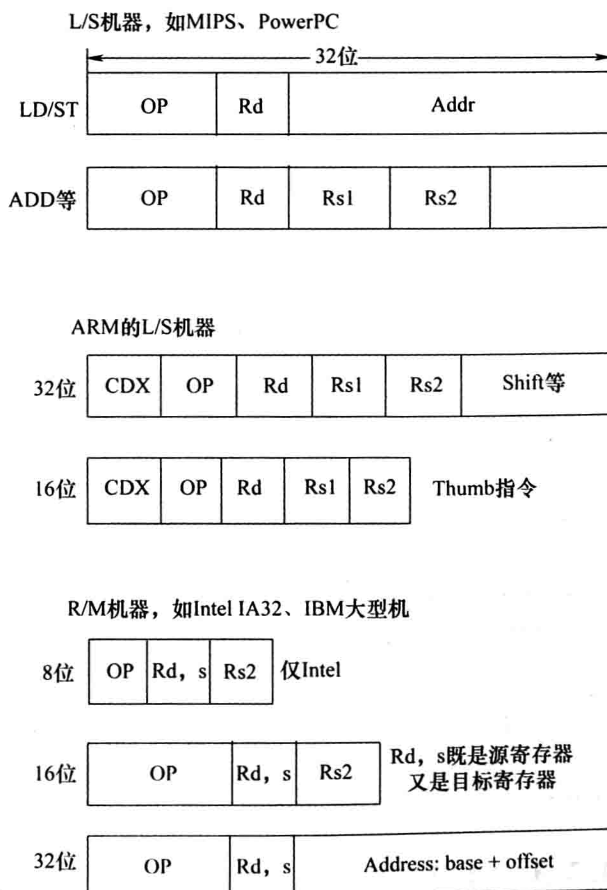


图3.5 一些典型处理器的指令长度和格式

图3.5给出了一些典型处理器的指令长部和格式。RISC机器运用固定的32位的指令大小或者具备64位指令扩展格式的32位格式。Intel IA32和IBM System390（现在称为zSeries）大型机运用了多种长度的指令。Intel用8位、16位、32位的指令，IBM用16位、32位和48位的指令。受限的寄存器组大小为Intel字节大小的指令实现提供了可能性。大小的可变性和R/M格式提供了很好的代码

密度。但与此同时译码也变得越发复杂。基于 RISC 的 ARM 格式是一种有趣的折中方式，它提供了带有内部条件域的 32 位指令集，所以指令可以被条件执行。另外，它还提供了 16 位指令集（称为 thumb 指令）。这些使这一指令集既具备译码的高效性又具备代码的高密度性。

指令集扩展的近期发展将在本书第 6 章详细介绍。

3.3.2 一些指令集习惯

表 3.3 给出了指令集助记符操作，包括了一些基本指令操作和常用助记符表示。通常，不同的数据类型（整型和浮点型）会运用不同的指令。为了指明操作特定的数据类型，操作助记符会扩展出一个数据类型的指示部分，所以“OP. W”可能指示一个整数型的 OP 指令，而“OP. F”指示一个浮点型的操作。表 3.4 给出了典型的数据类型修饰后缀。一个典型的指令形式为“OP. M 目标，源操作数 1，源操作数 2”。源和目标所指定的形式为一个寄存器或者一个内存地址（通常指定为一个基地址加上一个偏移地址）。

表 3.3 指令集助记符操作

助记符	操 作	助记符	操 作
ADD	加法	STM	存储多个寄存器
SUB	减法	MOVE	传送（寄从寄存器到寄存器或从内存到内存）
MPY	乘法	SHL	左移
DIV	除法	SHR	右移
CMP	比较	BR	非条件分支
LD	加载（从内存到寄存器）	BC	条件分支
ST	存储（从寄存器到内存）	BAL	分支和连接
LDM	加载多个寄存器		

表 3.4 典型的数据类型修饰后缀（OP. modifier）

修饰后缀	数据类型	修饰后缀	数据类型
B	字节（8 位）	F	浮点（32 位）
H	半字（16 位）	D	双精度浮点（64 位）
W	字（32 位）	C	字符或 8 位形式的十进制数

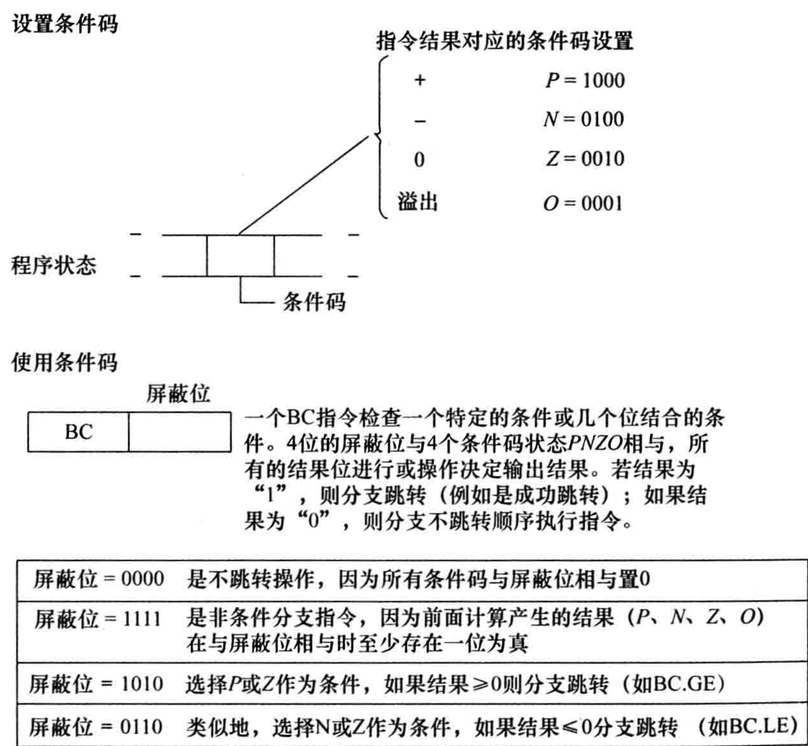
3.3.3 分支

分支（或跳转）管理程序的控制流，它们通常由非条件分支 BR、条件分支

BC 和子程序调用和返回（连接）组成。BC 指令检测条件码 CC 的状态，CC 的状态通常由程序状态或控制寄存器中的四位组成，通常而言，CC 由 ALU 指令设置来记录一系列结果（编码成2位或4位）。例如，以下4组可用来说明指令是否产生：

1. 一个正数结果。
2. 一个负数结果。
3. 一个零结果。
4. 一个溢出。

图3.6给出了使用条件码的 BC 指令实例。非条件转移 BR 所有掩码屏蔽位设置为“1”的条件转移方式，可以让每个指令包含一个屏蔽位来让所有指令可以条件执行（就像 ARM 指令集一样）。



屏蔽位 = 0000	是不跳转操作，因为所有条件码与屏蔽位相与置0
屏蔽位 = 1111	是非条件分支指令，因为前面计算产生的结果（P、N、Z、O）在与屏蔽位相与时至少存在一位为真
屏蔽位 = 1010	选择P或Z作为条件，如果结果≥0则分支跳转（如BC.GE）
屏蔽位 = 0110	类似地，选择N或Z作为条件，如果结果≤0分支跳转 （如BC.LE）

图3.6　使用条件码的 BC 指令实例

3.3.4　中断和异常

很多嵌入式 SoC 的控制器拥有外部中断和内部异常，这意味着它需要中断管理器（或处理器）程序的帮助，这些装置可以以如下的各种方式进行管理和支持：

1. 用户请求与被迫产生。前者经常涉及一个错误的操作，如除零操作；后者通常由外部事件引起，如设备错误。

2. 可屏蔽与不可屏蔽。前者可以通过设置中断屏蔽位屏蔽；后者不能屏蔽。

3. 中止与恢复。除零这样的操作事件会中止普通的处理过程，之后处理器可以恢复操作。

4. 异步与同步。当一个中断事件在程序的执行过程中触发时，中断事件可以由异步于处理器时钟的外部设备产生，也可以由与处理器时钟同步的时钟产生。

5. 指令间与指令内。中断事件可以产生于两条指令之间或者一条指令执行的过程中。

通常而言，上述两者的第一个选项容易被实现而且可能在当前指令完成之后被处理。设计者为了约束设计需要精确的例外。例外的精确的意思是，在例外发生前的所有指令都被正确完成，而例外发生后的所有指令都没有改变状态。当例外被处理结束后，后续的指令重新开始执行。

另外一些中断异常事件会同时产生甚至出现相互嵌套的形式，所以需要设置它们的优先级，控制器和通用处理器拥有特殊的部件来处理这些问题并保留系统的现场来继续执行异常。

3.4 处理器微体系结构的基本概念

几乎所有现代处理器都应用了一种指令执行流水级的设计。简单的处理器每个周期只发射一条指令，其他处理器可以发射多条。很多嵌入式和信号处理器运用简单的每周期发射单指令的设计方法，但是大多数现代台式机、笔记本式计算机和服务器都使用的是每周期多指令发射的方式。

每一个处理器（见图 3.7）具备内存系统、执行部件（数据通路）和指令部件。缓存和内存的速度越快，读取指令和数据（IF 和 DF）所需的周期就越少；运算部件越多，运算周期（EX）就越小。缓存和执行部件的控制是由指令部件完成的。

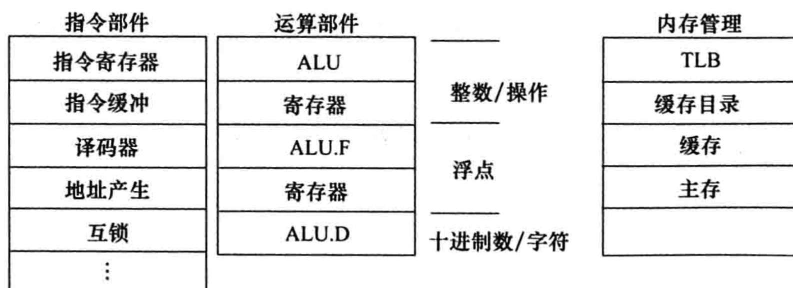


图 3.7 处理器部件

流水线的原理和控制有多种可能性：流水线可以在每一周期执行一条或多条指令；指令的译码和/或执行可能遵循程序次序也可能不遵循。事实上，在多线程流水线中不同程序的指令也可能在同一周期执行。表3.5给出了一些流水线处理器的类型。

表3.5 一些流水线处理器的类型

类　型	每周期译码指令数	注　释	通常情况相对性能
局部或静态流水线	1或更少	所有动作有序	0.5~0.9
典型流水线	1	所有D和所有WB有序	1.0
乱序（执行）流水线	1	所有D有序，WB无序	1.2
多发射超标量结构	4	没有顺序限制（只有依赖顺序）	2.5
多发射VLIW	8	编译器排序	3.0
多线程超标量结构	4	通常为2个线程	3.0

不管是什么类型的流水线处理器，“停顿（break）”或延迟是限制流水线性能的主要因素。

出现延迟或停顿主要由以下4种原因中的一种产生：

1. 数据冲突——源操作数不可用。这种情况可能由多种原因产生的。通常当前的指令需要的操作数是前面未完成的指令的结果。扩展操作数的缓冲可以使这种效应最小化。

2. 资源竞争。多个连续的指令共用同一资源，或者是执行时间较长的指令延误了后续指令的执行。增加资源（浮点部件、寄存器端口和乱序执行）可以减少竞争。

3. 运行延迟（仅在顺序执行中出现）。当指令必须按照程序顺序完成WB（写回）阶段时，任何在执行阶段（如乘法或除法指令）的延迟可能会推迟流水线中执行的指令。

4. 分支。流水线由于分支的结果和/或在流水线继续执行前读取分支目标指令而延迟。分支预测，分支表和缓冲可以用来将分支的效应最小化。

3.5节将关注简单流水线的控制和基本流水线的操作。这些简单的处理器有最小的复杂度，但是却会承受最多的突发事件（大多数由分支产生）。接下来将考虑在流水线中和部件之间负责管理数据传输的缓冲。由于最优的流水线布局（流水线级数）与停顿发生的频率密切相关，将关注分支和减小分支流水线延迟效应的技术；然后，关注多指令执行和更健全的流水线控制。表3.6给出了一些SoC处理器的特性。

表 3.6 一些 SoC 处理器的特性

SoC	指令集	类型	指令长度	扩展
Freescale e600[101]	PowerPC	L/S	32 位	矢量扩展
ClearSpeed CSX600[59]	专有指令集	L/S	32 位	SIMD 96 PE
PlayStation 2[147,187]	MIPS	L/S	32 位	矢量扩展
AMD Geode[14]	IA32	R/M	1 字节或更多	MMX，3DNow!

3.5 指令处理的基本元素

一个指令单元由指令定义的状态寄存器、指令寄存器、指令缓冲器、译码器和一个互锁部件组成。指令缓冲器的功能是将指令读取到寄存器中，使指令可以快速进入指定位置等待译码。译码器负责控制缓存、ALU 和寄存器等。在流水线系统中指令部件的排序通常由硬件严格管理，但是运算部件可能被微程序控制，所以每一个进入运行阶段的指令会有一个与自身相关的微指令。互锁部件的职责是保证并行执行的多条指令得到完全顺序执行一样的结果。

由于指令在各个阶段被执行，所以即便是简单的流水线处理器也有许多设计和权衡策略。

图 3.8 所示的指令部件展示了处理器控制或指令部件和与内存的基本交互通路。

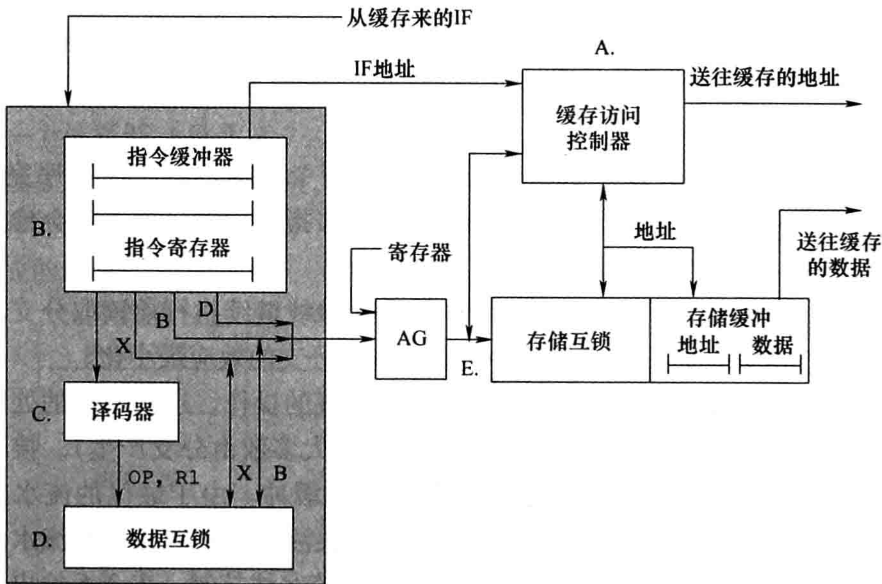


图 3.8 指令部件

3.5.1　指令译码器和互锁

当一条指令译码时，译码器必须为这条指令提供除了控制和序列信息外更多的信息。当前运行的指令是否合理，取决于流水线内的其他指令。译码器进行以下工作（见图3.9）：

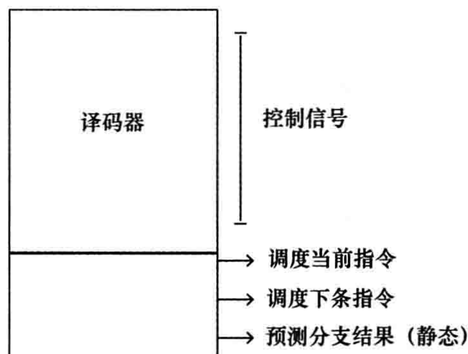


图3.9　译码器功能

1. 调度当前指令。当产生数据相关（如地址产生或AG周期）或产生例外（如不在旁路转换缓冲TLB中）和缓存失效时指令可能被延迟。

2. 调度后续指令。为了保证有序地完成，如当前指令有多个执行周期，后续的指令可能被延迟。

3. 在分支指令处选择（或预测）路径。

数据互锁（图3.8所示的组件D）可能是译码器的一部分，它决定寄存器的依赖关系，并完成对AG和EX部件的调度。互锁保证了当前指令不会用（依赖）一个前面指令产生的不可用的结果，直到这个结果可用为止。

执行控制器对后续的指令有相同的作用，确保在执行部件调度完成当前指令前，后续指令不会进入流水线，并且如果需要保持执行的顺序。

互锁的效果（见图3.10）在于每一条指令一旦开始译码，它的源寄存器（作为操作数或地址）必须与前面发射但没有完成的指令的目标寄存器比较

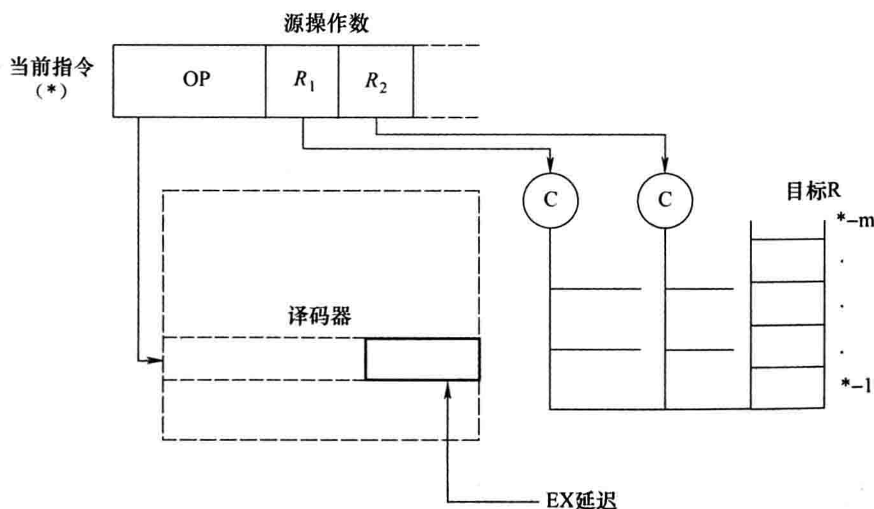


图3.10　互锁

(图3.10所示的C）来决定依赖关系。操作码本身通常确定了所需EX的周期数(图3.10所示EX框)。如果定时器检测的超过这个数目，后续的指令必须延迟保证执行次序。

存储互锁（E）针对存储地址与数据互锁的功能相同。在一个存储指令中，地址被发送至互锁部件，那么后续从AG（数据读请求）或IB（指令读请求）的读请求可以用于与挂起的存储操作比较并检测依赖关系。

3.5.2 旁路

旁路或前递是一种数据通路，它们将结果（通常来自于ALU）传送给一个使用者（可能仍然是ALU)，在此过程中绕过结果寄存器（后续被更新)。这种技术可以让ALU产生的结果在更早的流水级应用。

3.5.3 执行单元

如同缓存，执行单元（特别是浮点单元）在性能和面积方面显得尤为重要。事实上，一个简单的浮点部件可能会占据与一个基本整数处理器核（不算缓存）一样或更多的面积。在简单的顺序流水线中，执行延迟（运行时）是决定性能的重要因素，更健壮的流水线会在整数和浮点操作上运用相应更好的算数算法。表3.7给出了一些典型浮点部件的面积-时间权衡关系。

表3.7 一些浮点部件的面积-时间权衡关系

实　　现	字长/位	寄存器组	执行时间（加-乘-除）	流　水　线	部件面积/A
最小	32	4	3-8-30	否	25
典型	64	8～16	3-3-15	否	50
扩展算数	80	32	3-5-15	否	60
多发射	64～80	40以上	2-3-8	是	200以上

表中，字长指的是操作数的大小（指数和位数)，假设为IEEE 754标准格式。执行的时间为估计的总执行周期。流水线指吞吐量，即该实现方式每周期是否支持执行一条新的操作。最后一列是实现方式的部件估计需要的面积。

最小的实现可能只支持专用应用和32位操作数；典型的实现为一个支持64位操作数的简单流水线的浮点单元；高级的处理器支持扩展IEEE格式（80位)，它保证了中间计算的准确度；多发射实现是一种典型的无旁路实现，如果实现支持超过4发射，那么其大小很可能翻倍。

3.6 缓冲：让流水线延迟最小化

通过去除事件产生和输入数据被应用的时间之间的耦合性，缓冲可以改变指

令时序事件的发生，这使得处理器在不影响性能的前提下可以承受一定的延迟。缓冲保存数据等待其进入一个阶段，通过这种方式来实现延迟容忍。

缓冲可以利用平均请求率或最大请求率[113]来设计。对于前者，知道了请求的预期数目，就可以权衡缓冲的大小与发生溢出可能性的关系。溢出不会在CPU内部缓冲中出现，但是当缓冲已满且又来了一个新请求时，会产生“溢出”。此种情况会迫使处理器降速来等待直到有空闲的缓冲项。这样每当溢出状况发生，处理器流水线会停顿以使溢出的缓冲访问内存（或其他资源）。例如，存储缓存，通常就是运用平均请求率。

最大请求率被用于请求控制性能的资源，如在线（in-line）指令请求或视频缓冲的数据入口。在这种情况下，缓冲的大小要足够充足来匹配缓存或其他存储设备的处理器请求率。一个合理大小的缓冲可以使处理器在缓冲没有用完其条目的情况下继续以最大请求率访问指令或数据。

3.6.1 平均请求率缓冲

假设 q 是一个随机变量，描述对于一个资源的请求大小（挂起的请求数量）；Q 是它的平均分布；σ 是它的标准差。

Little **定理**：平均请求大小等于平均请求率（每周期的请求次数）乘以处理一个请求的平均时间[142]。

假设一个缓冲的大小是 BF，并且定义缓冲溢出的可能性为 p。根据马尔科夫不等式和切比雪夫不等式，p 有两个上界。

马尔科夫不等式

$$\text{Prob}\{q \geqslant \text{BF}\} \leqslant \frac{Q}{\text{BF}}$$

切比雪夫不等式

$$\text{Prob}\{q \geqslant \text{BF}\} \leqslant \frac{\sigma^2}{(\text{BF}-Q)^2}$$

运用两个不等式，对应给定溢出可能性 p，可以保守地选择 BF，两个不等式产生了两个上界，则

$$\text{BF} = \min\left(\frac{Q}{p},\ Q + \frac{\sigma}{\sqrt{p}}\right)$$

例 3.1 假设想要确定一个两项的写缓冲的效力。假设写请求率为每周期 0.15，完成一个存储操作的期望周期数是 2，应用 Little 定理，平均请求大小为 $0.15 \times 2 = 0.3$。假设 $\sigma^2 = 0.3$ 为请求大小，可以计算溢出可能性的上界为

$$p = \max\left(\frac{Q}{\text{BF}},\ \frac{\sigma^2}{(\text{BF}-Q)^2}\right) = 0.10$$

3.6.2 固定或最大请求率的缓冲设计

一个提供固定请求率的缓冲在概念上是容易设计的，主要的考虑在于屏蔽访问延迟。如果每1个周期处理一项而需要3周期去访问这个项目，那么需要1个至少三项的缓冲。如果将正在处理的项目考虑进去的话，则至少需要4项。

通常而言，最大请求率缓冲为了处理提供了一个数据或指令的固定率。这样的缓冲有很多，包括指令缓冲、视频缓冲、图形和多媒体缓冲。

一般情况下，当每周期处理 s 个项目，p 个项目用固定的访问时间（access time）从存储设备中取出时，缓冲大小 BF 为

$$\mathrm{BF} = 1 + \left\lceil s \times \frac{\text{access time(cycles)}}{p} \right\rceil$$

初始的“1”是当前周期用于处理的单入口缓冲的修正值，在某些情况下它可能并不需要。这里所指的缓冲是为了缓冲功能部件或译码器入口而设计的（即指令译码器）。它与帧缓冲和图像缓冲这种在处理器和媒体设备之间转换的缓冲并不一样，然而这里所述的缓冲的原则也同样适用于那些媒体缓冲。

3.7 分支：减少分支的开销

分支是优化处理器性能的难题之一。分支可以显著地降低性能，如一个条件分支指令（BC）需要检测前一条指令所设置的条件码，在分支指令译码和条件码设置之间可能相差很多拍，如图3.11所示。最简单的一种策略是处理器不做

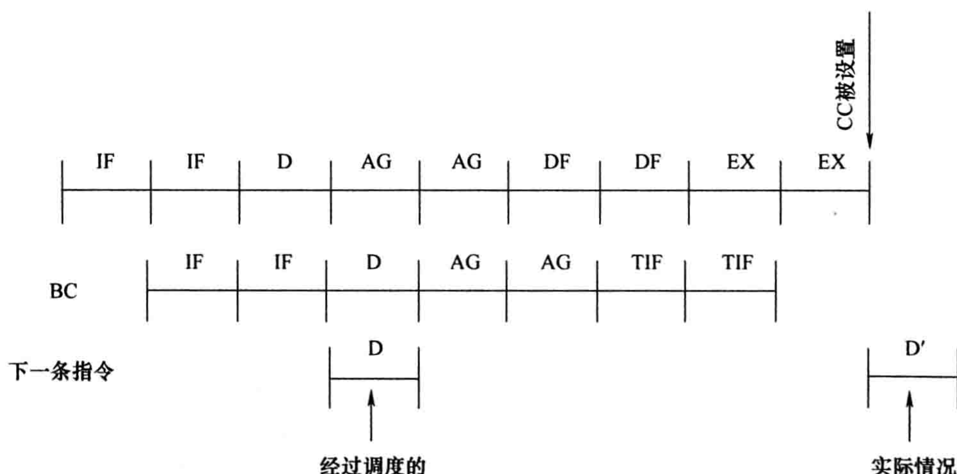


图3.11 由分支指令（BC）产生的延迟

任何操作，简单地等待条件码设置的输出结果并推迟条件分支指令后面指令的译码，直到条件码被算出。当分支指令结果为转移时，在原分配给算术指令取数的时间段获取分支目标。这个策略简单易于实现，并且使由分支指令引起的额外的内存数据传输最小化。尝试猜测特定路径的策略更加复杂，并会可能出现错误引发额外的取指。

如图3.11所示，实际的译码可能是5拍延迟（如一个5拍分支惩罚），而这并不是全部的副作用。当没有跳转到目标路径时，分支指令后的第2条指令的时序也会因为这条指令没有预取而延误额外的1拍。

由于分支是处理器性能的一个主要限制因素[60,219]，有很多研究致力于减少分支的负面效果。有两种简单的和两种实质性的处理分支问题的方法。两种简单的方法如下：

1. 消去分支。对于确定的代码序列，可以将分支替换成其他的操作。

2. 简单的分支加速。这种方式减少了目标指令的取值和条件码决策所需要的时间。

两种更复杂的方法是这两种简单方法的一般化：

1. 分支目标获取。在一个分支指令执行之后，可以将它的目标指令（和它的地址）存在一张表里，为在之后的运行中避免分支延迟。如果可以预测分支的路径的结果并且已有缓冲中的目标地址，那么分支指令的执行将不会产生延迟。

2. 分支预测。可以通过分支指令的相关可用信息来预测分支的结果并开始执行预测的程序路径。如果这个策略是简单或是细微的，总是取跳转条件分支的方向，这种策略被称为固定策略；如果这个策略由操作码的类型或目标方向决定，则这种策略被称为静态策略；如果这个策略由当前程序行为决定，则这种策略被称为动态策略（见图3.12）。

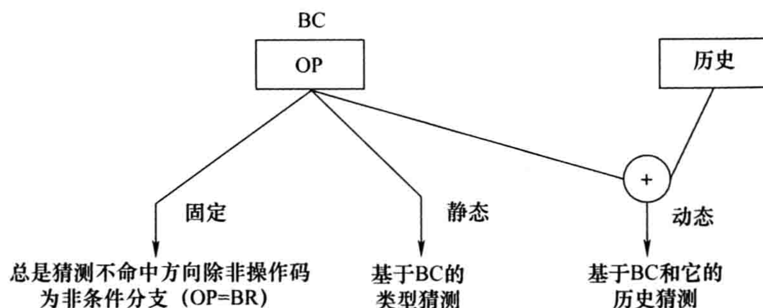


图3.12 分支预测

分支管理技术见表3.8后面将会主要关注上述两种一般方法。

表 3.8 分支管理技术

方法	做法	硬件消耗	分支延迟的效果（跳转分支）	分支预测的效果
分支解决				
较早的条件码设置	较早地决定条件码的结果	无	可以节省一个周期	无
延迟的分支	较早地决定条件码的结果	无	可以节省一个周期	无
分支加法器	较早地决定目标地址	无	通常节省一个周期	无
减少分支目标延迟				
分支表缓冲	为每个分支指令存储最近目标指令在一个特殊的表中（BTB）	表可能较大	减为零	80% ~90% 以上的命中率，取决于大小和应用
提高分支预测率				
静态方式	运用分支指令操作码或测试码预测结果	小	无	70% ~80% 的准确度
三种动态技术				
双峰形	记录每个分支的结果	小表	无	80% ~90% 的准确度
两级适应	创建分支结果的矢量	可以是 16KB 以上	支持路径推测	95% 以上的准确度
双峰和两级适应结合方法	运用最好的结果	以上的所有	以上的所有	以上的所有

3.7.1 分支目标获取：分支目标缓冲

分支目标缓冲（Branch Target Buffer，BTB）存储了前面执行的分支指令的目标指令，如图3.13所示。每一个BTB项有一个当前指令地址（仅当分支出现别名情况时需要）、分支目标地址和最近目标指令。因为不需要等待地址产生的完成，目标地址可以使流水线中的目标取指的初始化更快。BTB的功能：每条指令在BTB索引，如果一条指令地址与BTB中的指令地址匹配，则做出一个预测判断这个地址的分支指令是否会跳转；如果预测结果为跳转会发生，那么目标指令会作为下一条指令；当分支指令被处理完，在执行阶段，如果实际目标与存储的目标不同，BTB可以更新正确的目标信息。

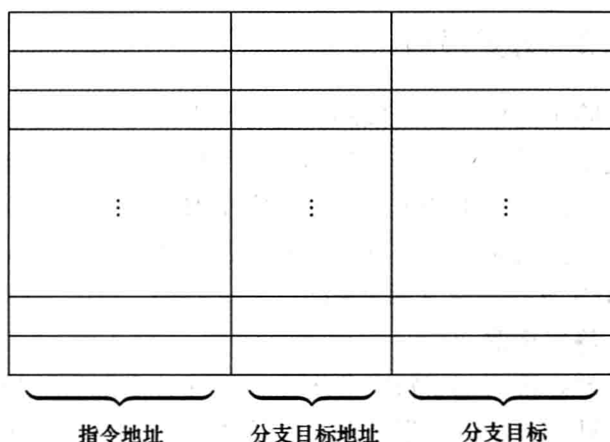


图3.13 分支目标缓冲［BTB有指令位索引，特定的分支可以通过引用表中的指令地址域来确定（避免别名冲突）］

BTB的效果取决于它的命中率——当分支指令被读取时它能在BTB中被查找到的可能性。512项的BTB的命中率在70%～98%之间，这取决于不同的应用。

BTB可以与指令缓存结合使用。假如有一个图3.14所示的配置，IF阶段可以用到BTB和指令缓存，如果IF阶段BTB命中，那么先前存在BTB中的目标指令会取出并且在常规调度时间送至处理器，处理器在执行目标指令时将不会产生分支延迟。

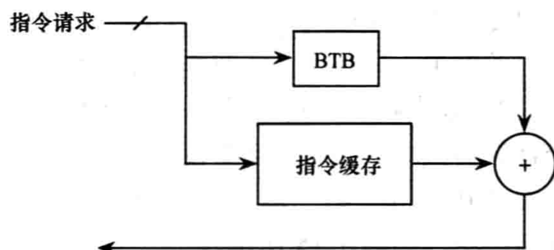


图3.14 典型的BTB结构（如果BTB“命中”，那么BTB向处理器返回目标指令，CPU猜测目标；如果BTB“不命中”，那么缓存返回分支指令和顺序路径，CPU猜测顺序路径）

BTB 提供了目标指令和新的 PC，只要预测正确，那么分支指令就没有延迟。需要注意分支指令本身仍然需要从指令缓存中读取并被完全执行。如果 AG 结果和 CC 的结果与预期不一致，那么所有从目标路径取出的指令都必须终止。很明显，条件执行（目标路径）指令都不能进行写结果操作，这样如果预测失败可以恢复。

3.7.2 分支预测

除了简单的固定预测，还有两类猜测分支是否跳转的策略：静态策略，基于分支指令的类型；动态预测，基于最近的历史和分支活动。

即便是完美的预测也无法完全取消分支延迟。完美的预测仅仅将条件分支的延迟转换成非条件分支的延迟，所以在运用更稳定（和昂贵）的预测器之前使用 BTB 支持是非常重要的。

静态预测

静态预测基于特定分支操作码或分支目标的相关方向。当一个分支指令译码时，针对分支的结果猜测，如果决定分支跳转成功，那么流水线取目标指令流并开始从此译码。一个静态分支预测策略如表 3.9 所示。

表 3.9 一个静态分支预测策略

指令类	指令	预测成功（S）	预测失败（U）
非条件分支	BR	总是	从不
条件分支	BC	反向猜测 S*	正向猜测 U*
循环控制	BCT	总是	从不
调用/返回	BAL	总是	从不

*当分支目标小于当前 PC 时，假设这是一个循环，跳转到目标地址，否则预测顺序执行。

表 3.9 中描述的策略的总体命中率的总体准确率为 70% ~80%。

动态预测：双峰形

动态策略根据历史来预测，也就是通过一个分支指令过去行为的序列——跳转还是不跳转——来决定。表 3.10 中，Lee 和 Smith[148] 给出了当预测基于前面运行分支指令的结果时的效果。预测的算法非常简单，在策略实现中运用一个小的上/下饱和计数器，如果分支跳转，这个计数器增加且有一个最大值 n，一个不跳转的分支使这个计数器减小。对于一个 2 位的计数器，“00”和“01”状态下预测分支不跳转，在“10”和“11”状态下预测分支跳转。这个表可以以分离或综合的形式存在缓存中，如图 3.15 所示。

表 3.10 n 位计数器预测准确率[148]

n	编译器	商务	科学	高级
0	64.1	64.4	70.4	54.0

（续）

n	编 译 器	商 务	科 学	高 级
1	91.9	95.2	86.6	79.7
2	93.3	96.5	90.8	83.4
3	93.7	96.6	91.0	83.5
4	94.5	96.8	91.8	83.7
5	94.7	97.0	92.0	83.9

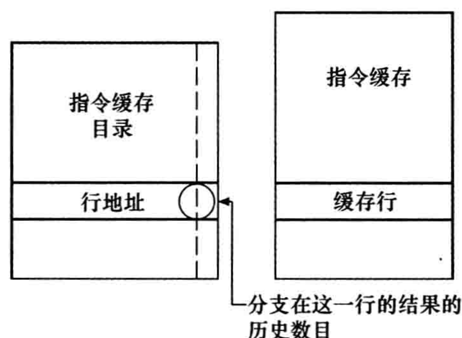


图 3.15 分支历史计数器可以存在指令缓存（上面）或在分开的表中

由于表的组织结构，两个不同的分支指令可能对应相同的历史，这会造成别名问题。

大量的研究表明（见表 3.10）。第一，预测的准确率随着位数的增加，增加的非常缓慢；第二，两位计数器的准确率为 83.4% ~96.5%，较表 3.9 中一位分支操作码的准确率高很多；第三，标准测试集（SPECmarks）的预测准确率在运用大型表的情况下为 93.5%。

动态预测： 两级适应

在多种环境中双峰型预测的预测准确率被限制在 90% 左右。Yeh 和 Patt[267,261] 运用适应型分支预测将预测准确率提升到 95%。基本的方法是为每一个分支指令关联一个移位寄存器，如一个分支缓冲表。这个移位寄存器记录分支的历史，例如，一个分支指令两次跳转，两次不跳转，那么移位寄存器记录为“1100”。每一种移位寄存器的状态当做地址索引一组计数器，如 2 位饱和计数器。每当移位寄存器遇到“1100”的形式，那么预测结果被记录到相应的饱和计数器中。如果分支跳转，计数器增加；如果分支没跳转，计数器减小。

适应性技术可能需要大量的硬件支持，需要与可能分支入口相关的历史位，同时还有存放预测结果的计数器表。大型程序中可能建立稳定的历史模式，所以

这种方法在大型程序中更为有效。

Yeh 和 Patt 的平均测试数据显示 6 位索引的适应性策略的预测准确率可以达到 92%，24 位索引的可以达到 95%。值得注意的是，SPECmark 的性能明显高于其他数据。

2 位饱和计数器在所有程序的平均预测准确率可达到 89.3%，然而图 3.16 的数据与表 3.10 中的数据基于不同程序集。

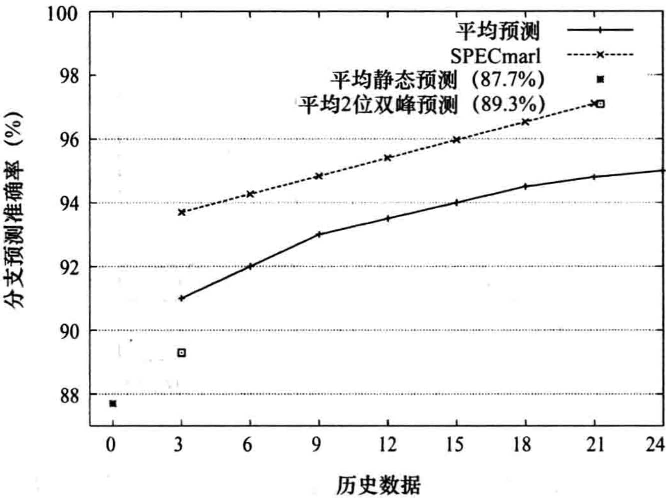


图 3.16　两级适应预测器的分支预测率

适应性预测在各种程序中预测准确率结果在本书参考文献[267]中给出。89% 和 95% 的预测准确率看起来没有那么明显，但是错误的分支预测延迟占据大部分的总体的运行延迟。

动态预测：　综合方法

双峰型和适应型方法提供了关于一个分支指令路径可能性不同的信息，因此可以通过增加另一个（2 位饱和）计数器的（投票）表将两种方法结合。当分支结果不同时，投票表在两种方法和最终结果中选择来更新投票表的计数。这就是综合方法，它能使预测准确率得到相应提高。当然也可以通过结合更多的预测方法来构建更稳定的预测器。

两级方式的缺点在于控制逻辑和两个序列表访问所需要的硬件开销。一种近似的方法称为全局适应预测器，它对于所有分支指令只使用一个移位寄存器来索引一个单一历史表，虽然它比两级方式快，但预测准确率只与双峰型预测器相当。但是可以将双峰型预测器和全局预测器结合创建一种近似综合方法，这种方法的结果可以与两级适应型预测器的效果相当。

表 3.11 给出了一些典型处理器的分支预测策略。表 3.12 中给出了一些 SoC 处理器分支预测策略，它们的策略明显比工作站处理器简单。

表 3.11 一些典型处理器的分支预测策略

工作站处理器	预 测 方 法	目 标 位 置
AMD	双峰型：16K×2 位	BTB：2K 项
IBM G5	3 表结合的方法	BTB
Intel Itanium	两级适应	目标在带有分支的指令缓存中
SoC 处理器	预测方法	目标位置
Intel XScale（ARM v5）	历史位	BTB：128 项

表 3.12 一些 SoC 处理器的分支预测策略

SoC	策 略	BTB 项数	分支历史项数
Freescale e600[101]	动态	128	2K
MIPS 74K[183]	动态	—	3×256
Intel PXA27x[132]	动态	128	—
ARC 600[19]	静态	—	—

3.8 更健壮的处理器：矢量、超长指令字和超标量体系结构

为了超越每条指令一个周期，处理器必须能够同时执行多条指令。并行处理器必须可以同时访问指令和数据内存并同时执行多项操作。能实现高度并发的处理器称为并行处理器，全称为具备指令级并行的处理器。

目前仅关注了执行单程序流的处理器，它们仅有单一的指令计数器所以是单处理器。但是通过指令顺序重排而不同于原来的程序顺序，并发指令的执行就可以实现了。

并行处理器比简单流水线的处理器更为复杂，对于这些处理器，性能在很大程度上取决于编译器的能力、执行资源和内存系统的设计。并行处理器依赖编译器检测程序中的指令级并行，编译器必须将代码重组成能让处理器并行处理的形式；并行处理器需要额外的执行资源，如加法器、乘法器和一个高级的内存系统提供以期望速度执行程序所需要的操作数和指令带宽[200,248]。

3.9 矢量处理器和矢量指令扩展

矢量指令可通过以下方式提升性能：

1. 减少执行程序所需的指令数（减少指令带宽）。
2. 将数据组织成有规律的序列，使硬件可以高效地处理。
3. 简化循环结构，因此消除循环执行的控制开销。

矢量操作需要扩展指令集，同时（为了更好的性能）扩展功能部件、寄存

器堆特别是系统的内存。

矢量通常源于大型数据数组，是传统数据缓存不能很好管理的一种数据结构，访问数组元素被一段地址距离（称为步幅）分开，会装填一段时间局部性很小的数据进入数据缓存。这样在这段数据被替换之前就不会重复使用（见图3.17）。

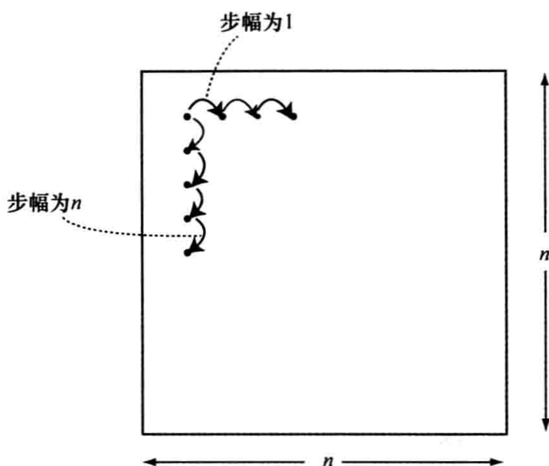


图3.17　对于内存中的不同数组，访问内存时不同的访问方式运用不同的步幅

矢量处理器通常包含矢量寄存器（Vector Register，VR）硬件来消去内存计算过程的耦合性。VR组是所有矢量操作数的源和目标，在许多实现中，访问绕过缓存，所以缓存只包含标量数据，标量数据不会在VR中使用（见图3.18）。

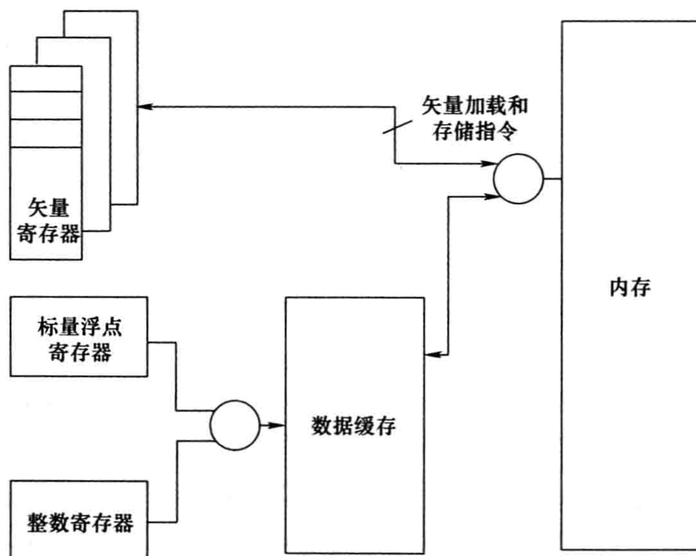


图3.18　矢量处理器中基本存储设备（矢量LD/ST通常绕过数据缓存）

3.9.1　矢量功能部件

VR通常由8个或者更多寄存器组组成，每个寄存器包含16～64个矢量元素，而每一个矢量元素是一个浮点字。

VR通过特殊的加载和存储指令访问内存，矢量执行部件通常为每一个指令类安排成独立的功能部件，可能包括以下几种：

- 加法/减法；

- 乘法；
- 除法或倒数操作；
- 逻辑运算（包含比较）。

由于矢量的目的是管理一组操作数的操作，一旦矢量操作开始，它可以以系统频率继续执形。图3.19给出了一个4级功能流水线的近似时序。一个矢量加（VADD）序列穿过了加法器的各个阶段，在四个加法阶段之后第一个元素的VR1和VR2（标为VR1.1和VR2.1）的和存于VR3（实际上是VR3.1）中。

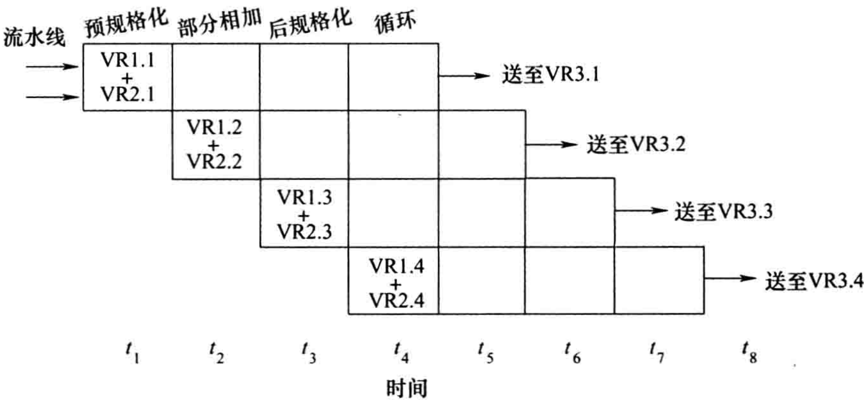


图3.19 一个4级功能流水线的近似时序

功能部件的流水线对矢量功能部件比对标量功能部件更重要。延迟对标量功能部件是最重要的。

矢量处理的优势在于需要很少的指令来执行矢量操作。一个单（重叠）矢量加载操作将信息存入VR。矢量操作以系统的时钟频率执行（每执行一个操作需要一个周期），一个重叠的存储操作与后续指令的操作重叠进行（见图3.20）。

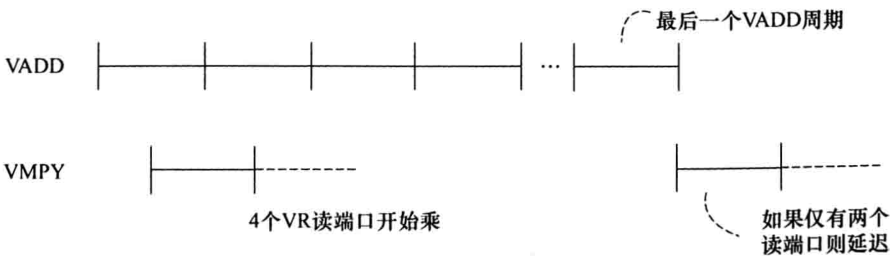


图3.20 VADD和VMPY示例（对于逻辑上独立的矢量指令而言，VR组的访问路径数目和矢量部件可能会限制性能。如果有四个读端口，VMPY可以在第二个周期开始，否则如果有两个读端口，那么VMPY必须等到VADD用完读端口后才开始）

矢量加载（Vector，Load，VLD）必须完成后才可以被使用（见图3.21），否则处理器不得不识别内存系统中操作数延迟的时刻。

示例
VADD V1，source(n)
VADD V2，V3，V4
VADD V5，V1，V6

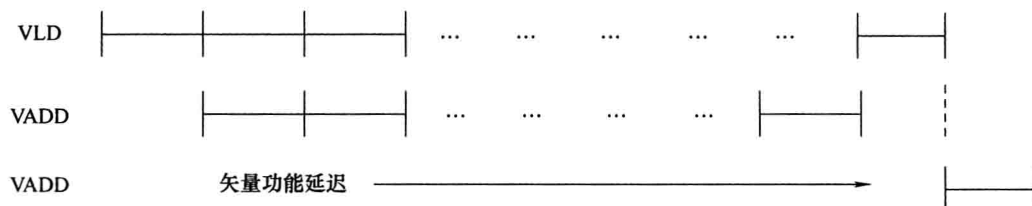


图3.21 VLD和VADD示例［虽然独立的VLD和VADD可以并行的进行（因为有充足的VR端口），但运用VLD结果的操作直到VLD完成才能开始］

处理器并行执行多个（独立的）矢量指令的能力也受到VR端口和矢量执行单元数目的限制。每一个并行矢量加载和存储都需要一个VR端口，矢量ALU操作需要多个端口。

在一些情况下，每周期执行多于一条矢量计算运算操作是可能的。通过旁路，一个矢量计算操作的结果可以直接用于后续矢量操作的操作数而不用先通过VR。图3.22和图3.23所示的这种操作称为链接。如图3.22所示，ADD-MPY链每个功能部件有四个阶段，如果ADD-MPY没有链接，那么每个指令需要4(启动)+64(元素/VR)=68个周期，也就是需要136个周期。在有链接的情况下，周期数减少到4(加启动)+4(乘启动)+64(元素/VR)=72个周期。

对于这两条指令
VADD VR3, VR1, VR2
VMPY VR5, VR3, VR4

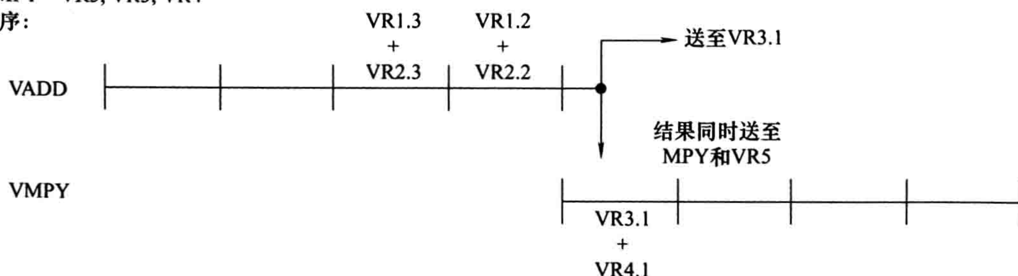


图3.22 矢量链的影响

管理内存访问是挖掘矢量处理器潜在性能的一个重要部分。由于算术操作每秒完成一次，运算代码重复地访问内存将新的矢量写入VR并将结果写入内存。一般来说，内存必须具备足够的带宽来支持至少每周期两字的频率（一个读一个写），甚至每周期3字（两个读和一个写）。这样的带宽允许两个矢量读和一

个矢量写的初始化和运行并行执行，同时执行一个矢量计算操作。如果内存到 VR 的内存带宽不充足，处理器在一个矢量操作完成后需要闲置，直到矢量加载和存储完成。处理器设计者要让一个矢量处理扩展适应标量设计（见表3.13），特别是内存系统，要适应快速矢量执行的需求，而不是简单地将标量处理扩展移植到标量处理器设计中。这是一个巨大的挑战。如果内存系统的带宽不够充足，矢量处理硬件所带来的性能提升会相应减少。

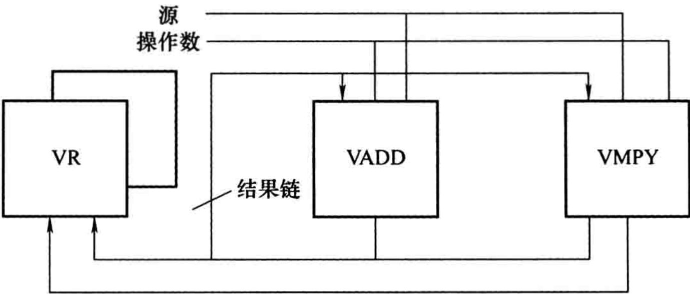


图3.23 矢量链通路

表3.13 内存可能需求（访问/处理器周期数）

	指 令	数 据
标量部件	1.0⁻*	1.0*
矢量部件	0.0⁺①	2.0~3.0②

* 通用的；可通过指令缓冲、指令缓存减少。

① 与其他需求相比相对小。

② 最少需要一个矢量加载（VLD）和一个矢量存储（VST）并行，两个 VLD 和 VST 并行更合适。

图3.24给出了通用矢量处理器的主要数据通路，展示了矢量处理器的主要

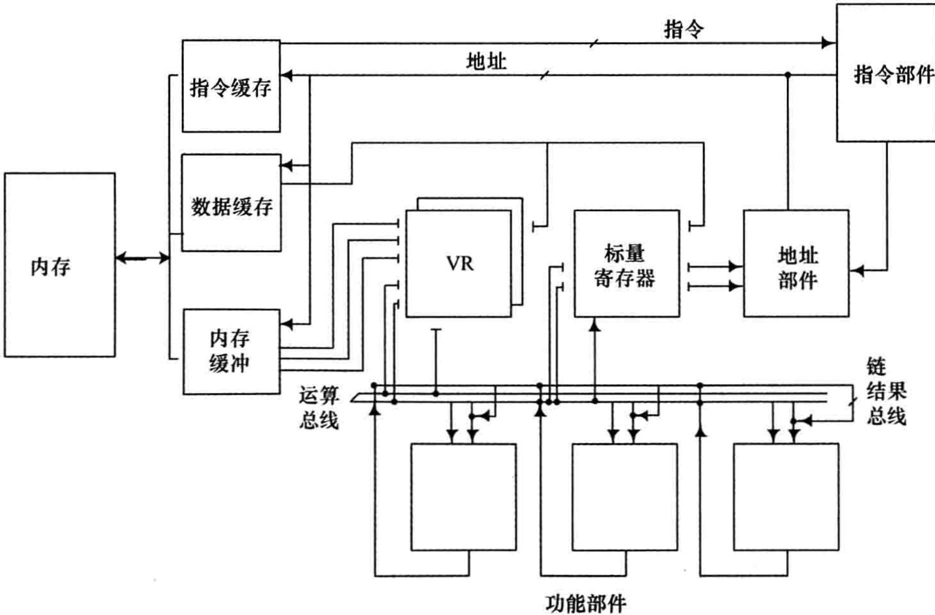


图3.24 通用矢量处理器的主要数据通路

元素。功能部件（加、减等）和两个寄存器组（矢量和标量或者通用寄存器）通过一个或更多的总线连接。如果允许链接（见图3.23），那么3个（或更多）操作数访问会同时从VR发出并且结果传回VR。另一个总线连接VR和内存缓冲。系统的其他部分指令缓存、数据缓存、通用寄存器等与常见流水线处理器中的一致。

3.10 超长指令字处理器

多发射处理器有两种广义分类：静态调度和动态调度。大体上这两种比较类似。指令组之间的依赖关系将被评估，不存在依赖关系的组将同时调度到多个执行单元。对于静态调度处理器，这一检测过程由编译器完成，指令被组装成指令包，在运行时被译码和执行。对于动态调度的处理器，独立指令的检测同样可能在编译时进行，代码可以被合理地编排成优化的执行形式，但是最终的指令选择（被执行或发送）由运行时译码器的硬件完成。原则上，动态调度处理器可能拥有的指令表示和形态上与较慢的流水线处理器没有显著差别。静态调度的处理器必须有一些额外的信息，明确或隐含地指示指令包的边界。

正如本书第1章提到的，美国Multiflow和Cydrome公司的处理器定义了早期VLIW机器[85]。这些机器所用的指令字包含10个指令段，每一段控制一个指定的执行单元，这样寄存器组有多个端口来支持对多种执行部件的同时访问。为了容纳多种指令段，指令字通常超过200位（见图3.25）。为了避免分支给性能带来的明显限制，一种新的称为跟踪调度的技术被开发出来。通过这种跟踪调度技术，分支的动态频率被大大降低。分支对可能之处进行预测，当达到可信成功率时被预测的路径成为一个更大基本块的一部分。这个过程一直进行，直到一个大小合适的块（没有分支的代码）可以被有效调度。如果在执行这段代码时出现了意料之外的（或不可预测的）分支，在基本块结束时，运用一个目标基本块修改正确的结果。

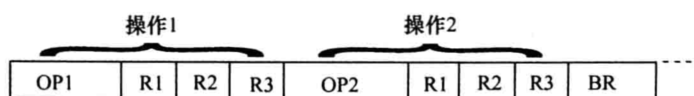


图3.25 部分VLIW格式（每一段并行地访问一个单一集中的寄存器组）

最近多发射处理器的研究方向倾向于更少的并行程度，然而同时多线程（Simultaneous Multithreading，SMT）的使用呈上升趋势。在SMT结构中，多个程序（线程）共用处理器执行硬件（加法器、译码器等），但拥有自己的寄存器组、指令计数器和指令寄存器。同一芯片上的两个2路SMT处理器（核）可以

支持4个程序同时运行。

图3.26所示为一个通用VLIW处理器的主要数据通路。寄存器端口的广泛运用为VLIW处理器提供了所需的同时访问数据的能力，这意味着寄存器组可能是处理器的瓶颈。

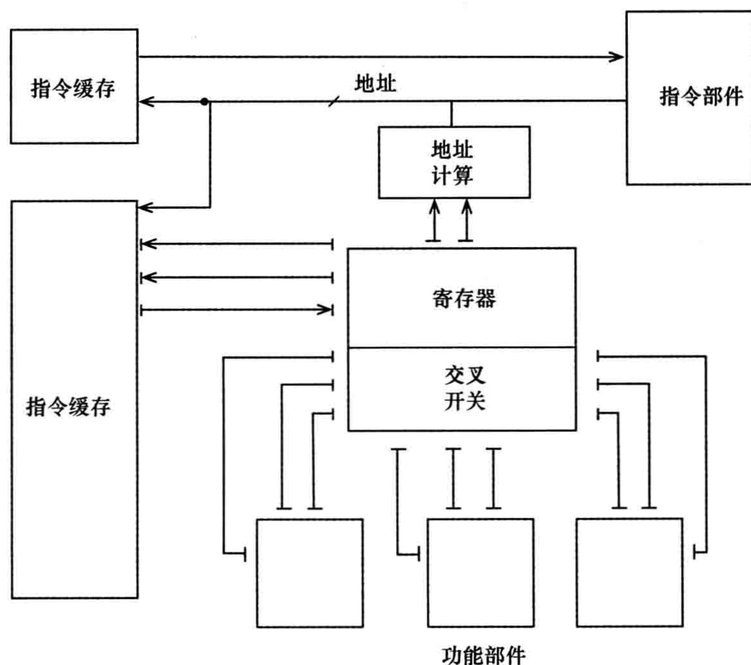


图3.26 一个通用VLIW处理器的主要数据通路

3.11 超标量处理器

超标量处理器也可以通过图3.26所示的数据通路方式实现。通常这种处理器运用多总线与寄存器组和功能部件连接，并且每一个总线服务于多个功能部件，这可能会限制最大并行度但可能减少需要的寄存器端口数。

不论检测过程是静态的还是动态的，指令内和指令间的相关性检测的问题在理论上是一样的（虽然在实际影响上不同）。下面将回顾指令独立性的理论。在超标量处理中，独立性检测必须在硬件执行，这必然使控制硬件和实现处理器的选择更加复杂化。本书还将讨论相对于其他方法更为具体和复杂的方法。

3.11.1 数据相关

在乱序执行中，两条指令I_i和I_j（i的执行顺序先于j）相关可能有三种。首

先是所谓的写后读（Read After Write，RAW）相关或真相关，当 I_i 的目标和 I_j 的源相同时出现，即

$$D_i = S_{1j} \text{或}$$
$$D_i = S_{2j}$$

这是一个数据或地址的相关。

另一种情况是当指令的目标与前一条指令的源相同时发生的相关，当以下情况时发生：

$$D_j = S_{1i} \text{或}$$
$$D_j = S_{2i}$$

当指令序列中的一条指令被延迟，后续的指令被允许超前执行而改变了前面指令源寄存器的内容，这时发生上述相关，如下面的例子所示（R3 是目标寄存器）：

I_1	DIV	R3，	R1，	R2
I_2	ADD	R5，	R3，	R4
I_3	ADD	R3，	R6，	R7

其中，第二条指令被第一条指令的除法操作延迟；如果允许第三条指令在它的操作数可用时立即执行，那么可能改变第二条指令计算用的寄存器（R3）。这种相关称为读后写（Write After Read，WAR）相关或顺序相关，因为仅在允许乱序执行时才会发生。

最后一种情况为指令 I_i 的目标和指令 I_j 的目标相同，即

$$D_i = D_j$$

在这种情况下，指令 I_i 可能在指令 I_j 之后完成，这时寄存器中的结果是由指令 I_i 产生的而不是由 I_j 产生的。这种相关称为写后写（Write After Write，WAW）相关或输出相关，在某种程度上它是值得讨论的。如果指令 I_i 产生的结果在指令 I_j 产生新的同目的的结果之前没有被后面的指令用到，那么图 3.27所示的第一个指令 I_i 是不需要的。这种类型的相关通常可以由编译器优化消除，在这里不予讨论。下面举两个例子。

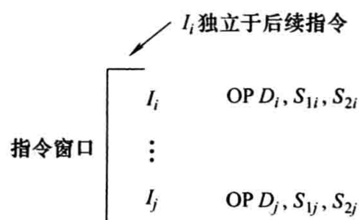


图 3.27 检测指令间得独立性

例 1

DIV	R3，	R1，	R2
ADD	R3，	R4，	R5

例2

DIV	R3,	R1,	R2
ADD	R5,	R3,	R4
ADD	R3,	R6,	R7

例1是冗余指令（DIV）的情况；而例2含有输出相关，但也有真相关，一旦真相关解决，输出相关也随之解决。代码中相关出现得越少，代码的并行性就越强，整个程序的执行也就越快。

3.11.2 检测指令并行

指令并行的检测可以在编译时、运行时或两者同时进行，很明显同时运用编译器和运行时硬件来支持并行指令运行能达到最好的效果。编译器可以进行循环展开并创建更大的基本块大小来减少分支。然而，完全的机器状态只有在运行时才能得到。例如一个由除、取数和除（divide，load，divide）组成的指令串，如果中间的加载指令产生了缓存失效，那么这个序列产生的明显的资源相关可能并不存在。

表3.14 一些SoC处理器的重命名特性

SoC	重命名缓冲大小	保留站数目
Freescale e600	16GPR，16FPR，16VR	8
MIPS 74K	32CB	—

注：GPR，通用寄存器；FPR，浮点寄存器；VR，矢量寄存器；CB，完成缓冲。

指令在译码时检查相关性，如果发现一个指令与其他前面的指令没有相关而且此时有可用的资源，那么这条指令将发射到功能部件。所有指令检查的数目决定指令窗口的大小（见图3.28），假设指令窗口有 N 条指令并在任意给定周期有 M 条指令被发射；在下一个周期，后续的 M 条指令进入缓冲，又有 N 条指令被检查，在一个周期内可能有 M 条指令被发射。

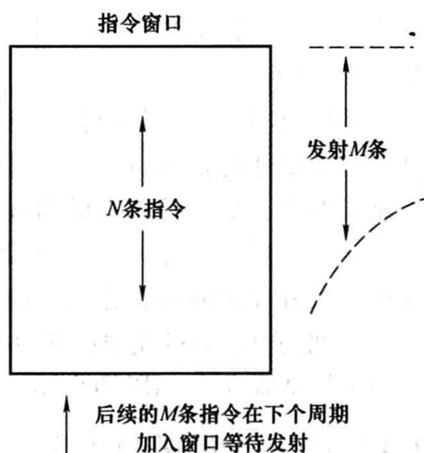


图3.28 指令窗口

顺序和输出相关可以用充足的寄存器来消除。当其中一种相关被检测时，重命名相关寄存器到另一个对指令集不可见的寄存器是可能的。这种重命名类型需要寄存器组扩展，包含重命名寄存器。一个常见的处理器可能由指令

集指定的 32 个寄存器组扩展为包含重命名寄存器的共 45 ~ 60 个寄存器的寄存器组（一些 SoC 处理器的重命名特性见表 3. 14）。

图 3. 29 给出了一个 M 级流水的处理器，它是检查 N 条指令并发射 M 条指令的整体布局。

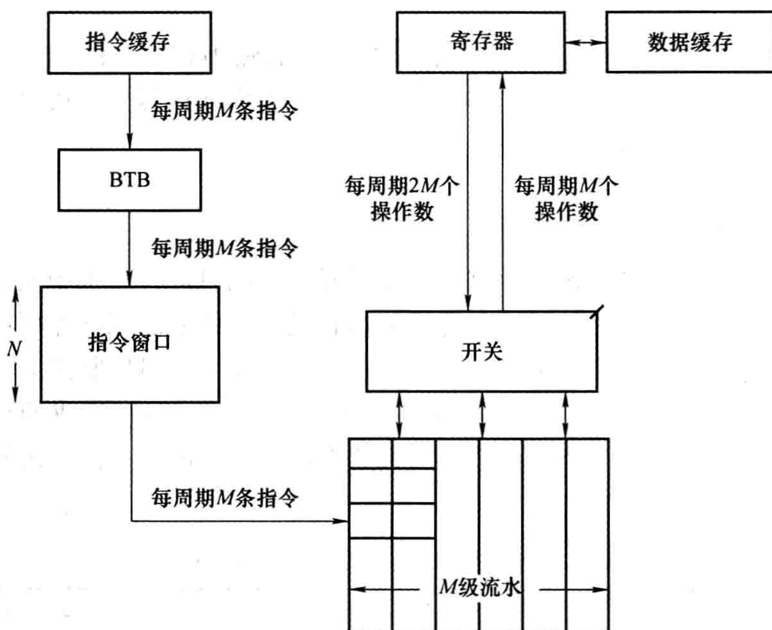


图 3. 29 一个 M 级流水处理器

在指令窗口里的 N 条指令都是发射的候选，发射与否取决于指令是否独立和执行资源是否可用。

如果处理器只支持两条 L/S 指令、一条浮点指令和一条定点指令。那么指令窗口的译码器必须选择这种 L/S 类型的指令发射。所以，即便三条 L/S 指令都是独立的也不能发射。

调度是将指定的资源在指定的时间分配给特定指令和它的操作数的过程，调度可以由功能部件本身在执行时通过集中或分散的方式来完成。前一种方式称为控制流调度，后一种方式成为数据流调度。在控制流调度中，在译码阶段解决相关，指令被保留（不发射）直到相关被解决。在数据流调度系统中，当指令被译码后离开译码阶段，在操作数和功能部件可用之前保留在缓冲中。

早期的机器通过控制流或数据流来确保指令乱序执行的正确性。CDC6600[238] 运用了控制流方法，IBM360 Model 91[244] 是第一个用数据流调度的系统。

3. 11. 3 一个简单的实现

本节观察一个简单的调度实现。虽然它的设置值 $N=1$、$M=1$，但它允许乱

序执行并说明了管理相关的基本策略。

考虑一个带有多种功能部件的系统，它的每一次执行可能需要多个周期，运用L/S结构，假设有一个集中的单一寄存器组为功能部件提供操作数。

假设最多N条指令已经被派遣执行，必须决定译码器当前如何发射一条指令。在还剩$N-1$条未发射指令的情况下发射一条单一的指令与在还剩N条指令的情况下发射是一样的。

运用一种有时被称为数据流方法或标签前递的方法，这种方法由Tomosulo[244]提出，并用他的名字命名。

寄存器组的每一个寄存器都被扩展，包含一个标签，它指示产生结果放入指定寄存器的功能部件。同样的，每一个功能部件有一个或者多个保留站（见图3.30）。

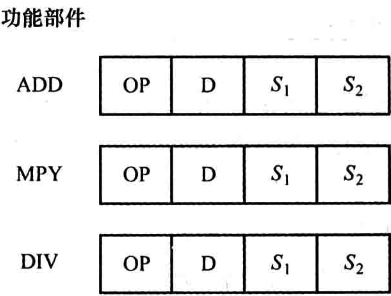


图3.30 保留站与功能部件相连（包括指令操作码和数据值或一个挂起状态等待进入功能部件相关的数据值；它们起到了重命名寄存器的作用）

保留站包含一个其他功能部件或者寄存器的标签，或者它可以包含需要的值。一个特定指令的操作数的值不需要对要发射至保留站的指令有效，一个值可能被一个特定寄存器的标签替代，直到这个值可用，保留站会一直等待。由于保留站保存了当前可用数据的值，它起到了重命名寄存器的作用，因此这个方案避免了顺序和输出相关。

控制分散在功能部件中，每一个保留站有效指定了自己的功能部件，一个浮点乘法器的两个保留站是不同的功能部件标签：乘法器1和乘法器2（见图3.31）。如果操作数可以直接进入乘法器，那么会有另一个标签：乘法器3。一旦一对操

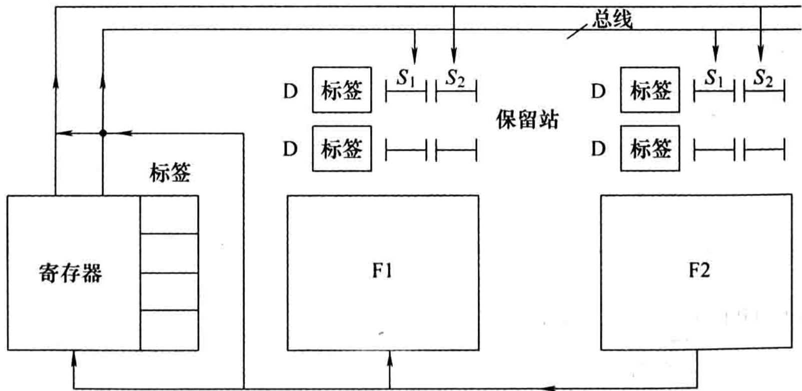


图3.31 数据流。每一个保留站由保存S_1和S_2的值（如果可用）的寄存器或指示值从哪里产生的标签组成）

作数有一个指定的功能部件标签，这个标签会一直伴随这个操作数对指导操作完成。任何取决于这个结果的部件（或寄存器）都要进行一个功能部件标签的复制，并且从总线上的广播得到结果。

对于前面的例子，有

DIV. F	R3,	R1,	R2
MPY. F	R5,	R3,	R4
ADD. F	R4,	R6,	R7

初始时 DIV. F 指令同 R1 和 R2 的值发射到除法部件（假设它们是可用的，从共用的总线中取到）。一个出发部件标签被发射到 R3，表示此刻它没有可用的值。在下一个周期，MPY. F 与 R4 的值和一个 R3 的 [DIV] 标签被发射到乘法部件。当除法部件完成时，它将自己的结果广播，因为它持有“出发部件”的标签，所以结果通知乘法部件的保留站。同时，加法部件同 R6、R7 中的值发射并开始加法，R4 从加法器处得到标签，由于乘法器已经拥有 R4 的旧值，所以没有顺序相关发生。

在数据流方法中，目标指向的寄存器的结果可能从来不实际到达这个寄存器。事实上，基于一个特定寄存器加载的计算可能会继续被传递给各种功能部件，所以在值保存前，一个基于新计算序列（一个新的加载指令）的新值可以运用这个目标寄存器。这种方法在一定程度上避免了中央寄存器组的应用，因此避免了寄存器顺序和输出相关。

对于顺序和输出相关是否是一个严重的问题存在一些争论[228]。当寄存器组更大时，一种优化编译器可以分配利用这些寄存器组，避免寄存器-资源相关的出现。当然所有的方案都没有解决真相关（第一种类型），然而大型寄存器组有它自身的缺陷，特别是由于中断引起的保存和还原时的通路问题。

研究 3.1　时序样例

对于代码序列

I_1	DIV. F	R3,	R1,	R2
I_2	MPY. F	R5,	R3,	R4
I_3	ADD. F	R4,	R6,	R7

假设执行时有三个独立的浮点部件：

除法	8 周期
乘法	4 周期
加法	3 周期

之后显示数据流的时序。

对于这种方法，可以得到以下分析结果：

第1个周期	译码器发射I_1→除法部件 R1→除法保留站 R2→除法保留站 TAG_DIV→R3
第2个周期	开始 DIV. F 指令 译码器发射I_2→乘法部件 TAG_DIV→乘法部件 R4→乘法保留站 TAG_MPY→R5
第3个周期	乘法器等待 译码器发射I_3→加法部件 R6→加法保留站 R7→加法保留站 TAG_ ADD→R4
第4个周期	开始 ADD. F 指令
第6个周期	加法部件请求下个周期广播（授权） 加法部件本周期完成
第7个周期	加法部件结果→R4
第9个周期	除法部件请求下个周期广播（授权） 除法部件本周期完成
第10个周期	除法部件→R3 除法部件→乘法部件
第11个周期	开始 MPY. F 指令
第14个周期	乘法完成并请求数据广播（授权）
第15个周期	乘法部件结果→R5

就目前的实现而言，发射逻辑被分散到保留站中，当同一周期多个指令将要被发射时，必须有多个分开的总线来传递信息：操作、标签/值1、标签/值2和目标。假设保留站与功能部件相连，如果为了实现方便集中保留站，那么设计将会与一个改进的控制流或计分板技术相似。

动作总结

可以总结出以下基本规则：

1. 如果数据值可用，译码器发射指令和数据值到保留站，否则发送指令和寄存器标签。

2. 目的寄存器（指令指定）获得功能部件标签。

3. 持续发射指令直到一种保留站满为止，未被发射的指令保持挂起状态。

4. 任何与未发射指令或挂起指令相关的指令必须保持挂起状态。

3.11.4 乱序指令的状态保存

乱序执行会导致机器状态明显的混乱，即使代码被正确地运行。如果发生了中断或者某种异常（甚至可能是一次错误的分支预测），判断异常的准确来源并明晰怎样为后续指令的处理保存和恢复机器状态会变得非常模糊不清。有以下两个基本方法可以解决这一问题：

1. 限制编程模型。这一方法只适用于中断并且会包含一个叫做非准确中断的设备，非准确中断仅说明一个例外在代码中的某个区域发生，而不进行更深的隔离。这种简单的方法可能适用于仅使用真实（没有虚拟）内存的信号或嵌入式设备，通常不适用于虚拟内存处理器。

如果一个加载指令访问到一个当前内存没有的页面而后续的指令已经在执行时，那将产生灾难性的后果。当缺失的页面被加载后控制交还给进程，这一加载过程可能与它相关的指令同时进行，但其他在这一加载指令前执行的指令不应被重复执行，所有的控制可能被搞乱。唯一的选择是在进程运行之前将这一进程需要的页面全部加载到内存中。对于实际可行的编程环境，如大型科学应用，这种方法是一种解决方案。

2. 创建一个写回阶段，保护寄存器组使用的顺序性，或者至少允许这样一个有序寄存器组的重构。

为了提供一个程序执行的顺序模型，必须有一些能够合理管理寄存器文件状态的机制，成功的关键[135,219]是有效地管理寄存器组和它的状态。如果指令有序执行，那么结果存在寄存器文件中（见图3.32），较早完成的指令需要保持挂起状态等待在它之前发射而未完成的指令，这样会牺牲性能。

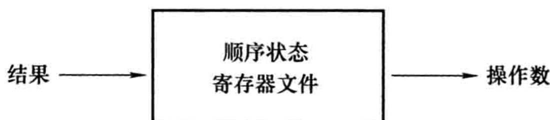


图3.32 简单寄存器文件的组织形式

另一种方法运用了重排序缓冲（见图3.33），结果乱序到达重排序缓冲，但是以程序顺序写回有序寄存器文件，这样一来保护了寄存器文件的状态。为了避免重排序缓冲的冲突，可以如图3.34所示的那样将缓冲分布到各个功能部件上。这些技术允许指令的乱序执行但是保护了寄存器组的有序写回。

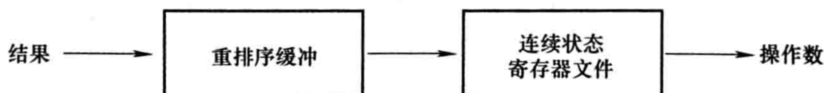


图3.33 集中的重排序缓冲方法

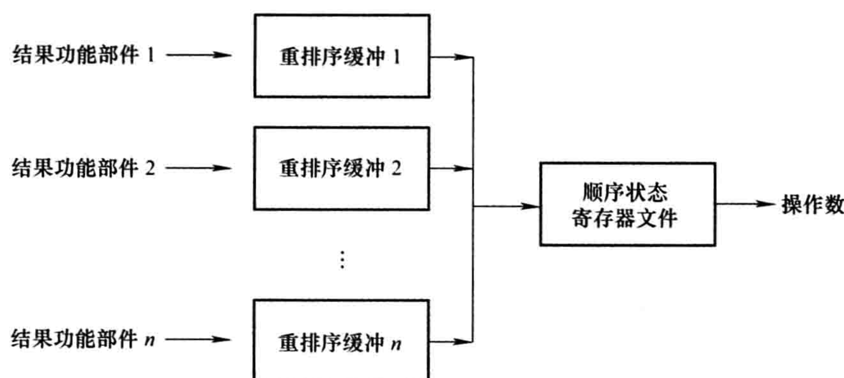


图 3.34　分散的重排序缓冲方法

3.12　处理器的演变和两个实例

下面来看看处理器的早期概念，然后看看最新的高性能处理器的实例。

3.12.1　软核和固核处理器设计：IP 形式的处理器

应用于 SoC 和其他特定应用领域的处理器，需要比通用处理器概念考虑更多。设计者仍然需要运用给定晶体管数目实现尽可能高的性能。目标是实现易于适用多种情况的高效的模块化设计，好的设计具备以下几点：

1. 一个有效运用数据指令内存（代码密度）和数据内存（一些操作数大小）的指令集。

2. 一个对于广泛的应用保持性能的有效微体系结构。

3. 一个相对简单、节约晶体管的结构。

4. 一定数目的协处理器扩展，易于添加到基本处理器中，可以包含浮点和矢量协处理器。

5. 对于所有处理器配置完全的软件支持，包括编译器和调试器。

ARM1020 处理器是这类处理器设计中的经典。为了改进代码密度，它运用了支持 16 位和 32 位指令的指令集，图 3.35 给出了 ARM1020T 处理器的数据通路。调试和系统控制协处理器和/或矢量和浮点协处理器可以直接添加来改进性能。ARM 总线也是一种 SoC 应用的标准。

其指令时序是简单的六级流水，如图 3.36 所示。由于结构简单，它的峰值性能可以达到每周期一条指令（忽略缓存失效）。

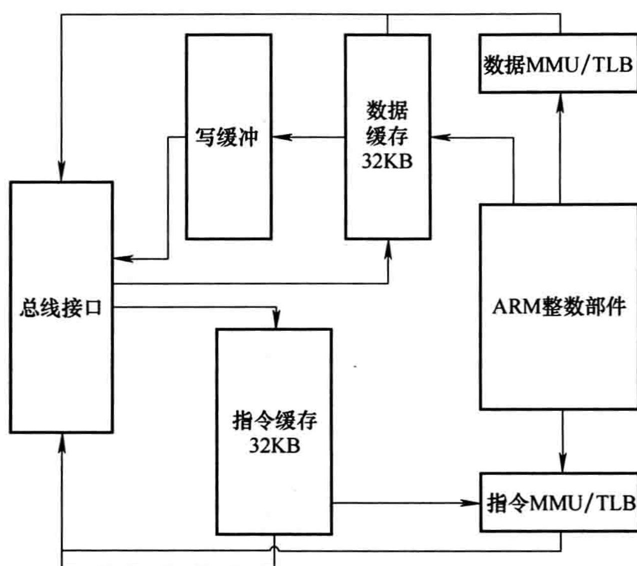


图 3.35　ARM1020T 处理器的数据通路[20]

整数流水线

IF1	IF2	ID	EX/AG	EX/DF	WB

图 3.36　ARM1020T 处理器指令时序

3.12.2　高性能定制处理器

当设计高性能工作站时，相对于性能，设计工作是次要的（但进入市场的时间更为重要）。这导致大型设计团队聚焦定制电路、时钟、算法和微结构，以此按计划达到性能要求。Freescale e600 处理器就是一个例子（见图 3.37）。这样的处理器运用了本章所述的所有设计技术，除此之外还有以下几种：

1. 由于有大量可用面积（晶体管），可以实现更多的分支表、多个执行部件、多发射和完全的指令乱序执行。

2. 时钟频率更高、周期时间缩短、一些基本的操作（如指令读取）占用多于一拍的时间，总之由于周期缩短，流水线变得更加复杂，时序图会明显变得更长（更多步骤）。

3. 因为容量大的缓存需要更长的访问时间，需要实现一层或多层的更大容量的缓存来支持小容量的一级缓存。

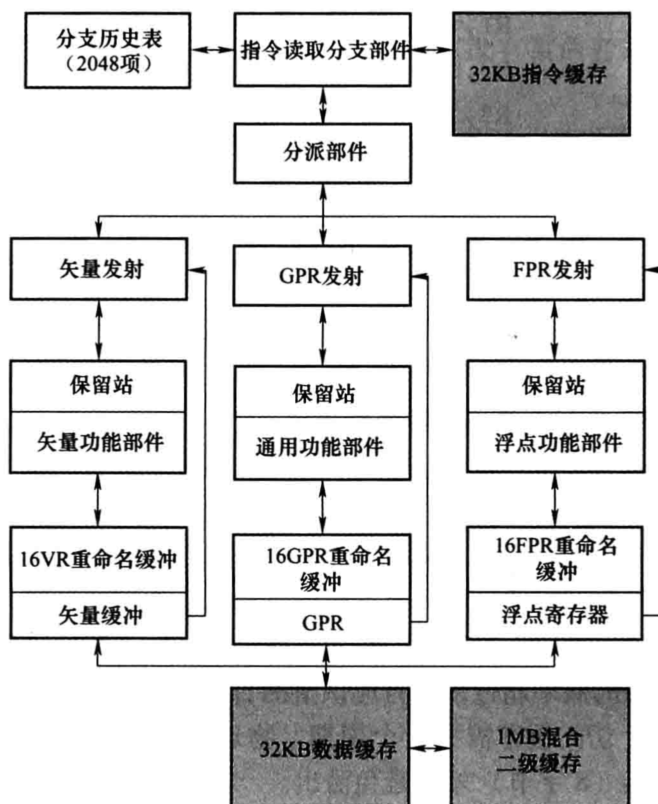


图 3.37　Freescale e600 处理器的数据通路[101]

3.13　总结

流水线处理器已经成为从大型机到微处理器几乎所有机器实现的选择。高密度 VLSI 逻辑技术，加上高密度内存让日趋复杂的处理器实现变成可能。

在模拟流水线处理器的性能时，为每条指令分配一个基本时间然后增加由代码运行产生的相关引发的延迟，这些相关通常由分支、数据依赖或有限的执行资源引发。对于每种类型的相关都有相应的实现策略缓解相关的影响。例如，实现分支预测策略，减缓了分支延迟的作用。然而，相关的检测由互锁来实现，互锁逻辑包括译码器检测相关并确保机器按照代码执行顺序合理地进行逻辑操作。

3.14　习题

1. 仿照研究 3.1，列出下面三条指令序列的时序：

ADD.F	R1,	R2,	R3
SUB.F	R3,	R4,	R5
MPY.F	R3,	R1,	R7

2. 在互联网上查找3种最新的处理器产品并列出它们的相关参数。

3. 假设一个矢量处理器在运行矢量代码时达到了2.5的加速比，在一个可矢量化代码率为50%的应用中，在一个非矢量的机器上的整体加速比是多少？对比一个每周期能最大运行4条算数操作的VLIW的期望加速比（VLIW和矢量处理器的时钟周期一致）。

4. 某一存储缓冲有4项，平均使用2项。

（a）在不知道方差的情况下，缓冲满或溢出的延迟可能性有多大？

（b）假设已知方差 $\sigma^2=0.5$，产生延迟的可能性是多少？

5.（a）假设某个处理器具有如下条件转移行为：猜对目标的时间开销为3个周期，如果猜错目标而代码实际上不跳转则有6个周期的惩罚；类似的，正确猜测不跳转没有延迟惩罚，但错误的猜测不跳转而实际上代码跳转则有6个周期的惩罚。程序跳转的概率超过多少时应该猜测目标路径？

（b）对于一个访问缓存需要3个周期，物理字长为8字节的L/S机器，指令缓冲需要多少字（8字节）才能避免溢出。

6.（a）当分支指令译码时访问一个分支表缓冲（Branch Table Buffer，BTB），这样在分支译码周期结束时目标地址（仅为目标地址）可用。

IF	IF	D	AG	AG	DF	DF	EX	EX

对于一个带有BTB的R/M机器，时序模板如上所示（每周期一个译码），非条件分支和条件分支的惩罚分别是多少？假设所有非条件分支和50%的条件分支在BTB中命中，80%的命中的条件指令实际上跳转了，而20%未命中的条件指令实际上跳转了。

（b）如果目标指令直接存放在BTB中，非条件分支和条件分支的惩罚分别是多少［假设条件同（a）］？

7. BTB可以和历史位一同决定什么时候将一个目标放入BTB中。这样小容量的BTB可以变得更高效。BTB大小低于多少时，2位分支历史方式更高效（在科学计算环境下）？

8. 寻找一种商业VLIW机器和它的指令布局，试着描述它，并写出计算 $A^2+7\times B+A\times C-D/(A\times B)$ 的指令序列，先将值加载到寄存器中再计算。

9. 重命名寄存器可以代替指令集指定的寄存器组。试着比较以下两种情况：没有寄存器组（在一个单一累加器的指令集中）；没有重命名寄存器，但在指令集中有一个大的寄存器组。

10. 寻找一个运用矢量处理器的 SoC 配置，从以下几方面描述矢量处理器的体系结构：寄存器组数量、每组的寄存器数、指令格式等。

11. 寻找一个运用超长指令字处理器的 SoC 配置，从以下几方面描述处理器的体系结构：寄存器组、重命名寄存器个数、控制流和数据流、指令格式等。

第4章 片上系统和基于主板系统的存储设计

4.1 引言

存储设计是系统设计的关键环节。存储系统通常是系统中消耗（面积或芯片数量）最大的部分，而且在很大程度上决定着系统的性能。除了处理器和互联结构，应用程序能运行的最大速度受到存储系统的限制，因为运行时所需要的指令和操作数都是由存储器提供的。

存储设计需要考虑很多方面的内容。首要方面就是应用程序的需求：操作系统、大小、应用程序的多样性等。这在很大程度上决定了存储器的大小和存储寻址方式：实地址或虚地址。图4.1所示存储设计框图。表4.1给出了不同存储技术所需空间对比。

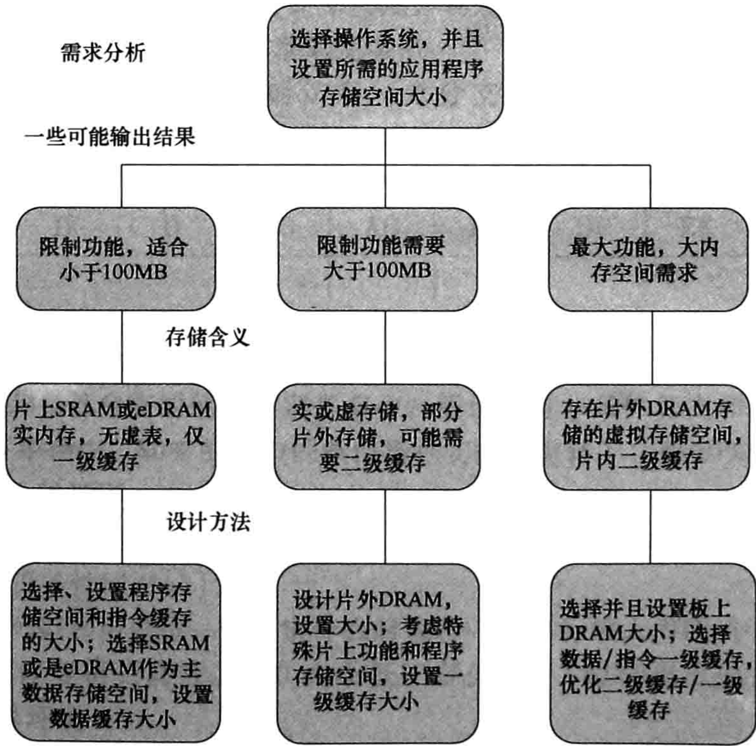


图4.1 存储设计框图

表 4.1　不同存储技术所需空间对比

存储技术	rbe	单位面积 A 上的存储空间/KB
DRAM	0.05 ~ 0.1	1800 ~ 3600
SRAM	0.6	300
ROM/PROM	0.2 ~ 0.8 +	225 ~ 900
eDRAM	0.15	1200
Flash：NAND	0.02	10000

首先看 SoC 外部存储和内部存储部分，然后分析暂存器和缓存来理解它们是如何工作和进行设计的。之后，我们将考虑主存问题，首先是片上存储器的设计，然后是传统动态 RAM（Dynamic RAM，DRAM）的设计。作为大型存储系统设计的一部分，还将会分析不同种类的存储模块、交叉存储技术和存储系统的性能等方面的内容。图 4.2 给出了一个 SoC 存储设计模型。在本章，互联网路、处理器和 I/O 设定为理想状态，这样就能够对存储器设计进行权衡考虑。

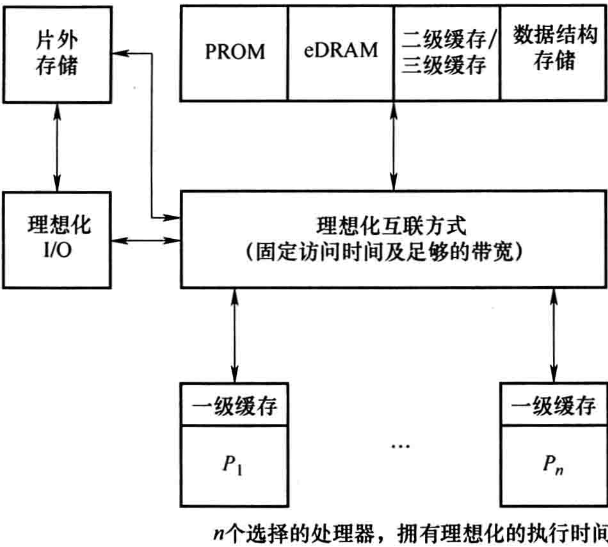


图 4.2　一个 SoC 存储设计模型

表 4.2 给出了一些闪存（NAND）的封装形式，它们是能集成到 SoC 设计的一些存储器。

表 4.2　一些闪存（NAND）封装形式（2 ~ 128GB）

形　　式	大小（近似值）/(mm × mm × mm)	重量/g	速度(读/写)/(MB/s)	典型应用程序
标准闪存(Compact Flash，CF)	36 × 43 × 3.3	11.4	22/18	数码相机
安全数位(Secure Digital，SD)	32 × 24 × 2.1	2.0	22/18	数码/摄像机
Mini SD	20 × 20 × 1.2	1.0	22/18	手机/GPS
Micro SD	15 × 11 × 0.7	0.5	22/15	迷你手机

例 1

需求的设计目标对能达到的性能值起到很大的作用。考虑图 4.1 所示两种不同的设计路线：最大功能设计方法和限制功能设计方法。无论是从片内还是片外存储器的设计上看，它们之间的区别是很细微的。然而，最终的性能可能大相径庭，因为片外存储器需要很长的访问时间。如果能够在片上存储器中获取内存内容（程序数据和代码），那么访存时间仅为 3 ~ 10 个时钟周期。

片外访问时间是一个更大的数量级（30 ~ 100 个周期）。为了达到相同的性能，片外存储设计需要增加大量的缓存，通常会将这些缓存划分为不同的层级来满足访问时间的要求。实际上，缓存的 1 位的面积比片内嵌入 DRAM（embedded DRAM，eDRAM）1 位大 50 倍（详见本书第 2 章和 4.13 节）。所以片外存储所需的更大缓存的开销可能达到 10 ×50 或 500 个 DRAM 位。如果一个存储系统使用了 10K 个动态电阻来组成缓冲用来支持片上存储，那么芯片需要 100K 个动态电阻来支持片外存储。其中 90K 个动态电阻的面积差别可以用来实现 450K 个 eDRAM 位。

4.2 概况

4.2.1 SoC 外部存储：闪存

闪存技术发展非常迅速，每隔一段时间就会有新的技术创新发布。闪存不仅是存储器的替代者，也被广泛认为是磁盘的替代者。然而，在一些环境和配置下，它能够同时达到内存和非易失性备份存储介质的作用。

闪存由一组浮栅晶体管组成，这些晶体管类似 MOS 晶体管，但是拥有独特的双门结构：控制门电路和绝缘浮栅门电路。电荷存储在浮栅端，提供非易失性存储功能。尽管数据能够被重写，但是目前的技术受限于可靠的重写次数，这个次数通常低于百万级。因为降级使用（degradation with use）会成为一个问题，所以需要频繁地进行错误检测和纠正。

对于半导体设备来说，尽管闪存在位密度方面有优势，但是其写周期太长的缺陷限制将其用于存储那些需要频繁修改的数据，如程序和大型文件。

目前为止，有两种闪存的类型结构：NOR 和 NAND。NOR 实现方式相对灵活，但是 NAND 却拥有更好的位存储密度。而 NOR/NAND 混合型实现则可能会将 NOR 阵列作为大块 NAND 阵列的缓冲器使用。表 4.3 给出了两种闪存类型的对比。

表 4.3 两种闪存类型的对比

技　术	NOR	NAND
比特密度/（KB/A）	1000	10000

（续）

技　　术	NOR	NAND
典型容量	64MB	16GB（可以由4块或是更多块组成）
访问时间	20~70ns	10μs
传输率/（MB/s）	150	300
写时间/μs	300	200
地址访问方式	字或块	块
应用程序	程序存储和 限制的数据存储	替代磁盘

闪存拥有各种封装形式。尺寸越大的通常越古老（见表4.2）。小的闪存片可以通过“堆叠”到SoC芯片上成为一个系统/存储器封装。闪存片还可以通过堆叠形成一个大型（64~256GB）存储器封装。

以目前的技术来看，闪存通常用于片外实现。然而，也有一些闪存变种设计用于和传统的SoC技术相兼容。例如，SONOS[201]是一种非易失性存储器，Z-RAM[91]是DRAM的一种替代品。这两者似乎都没有受到重写次数的限制。Z-RAM在提供更好的存储密度的同时，似乎通过其他方式与DRAM速度兼容了。而SONOS在提供更好的存储密度的同时，其访问时间比eDRAM短。

4.2.2 SoC内部存储器：放置点

存储系统设计中最重要也最显著的因素就是主存储器的放置点：片内（和处理器在同一块芯片上）和片外（独立芯片或是在多块芯片的某模块上）。就像本书第1章指出的情况一样，这个因素将传统的工作站处理器和基于应用的板卡设计同SoC设计区别开来。

存储系统的设计受到两个基本因素的限制，这两个基本因素决定着存储系统的性能。第一个因素是访问时间。访问时间是处理器的访存请求被传输到存储系统，访问数据单元，然后将数据返回给处理器的整个过程时间。访问时间很大程度地受到物理设计因素的限制，如处理器和存储器的物理距离或者是总线延迟、芯片延迟等一些其他因素。第二个因素是访存宽度，即单位时间内存储器对访问请求的反应能力。带宽主要是由物理存储器系统的组织方式——独立存储器阵列的数量及特殊顺序访问模式的使用——所决定的。

缓存系统必须弥补访存时间和带宽的限制。

目标定位于高性能的工作站处理器需要非常高效的存储系统，如果把存储器放置在片外就很难实现这一点。表4.4给出了不同环境下存储系统设计方法的对比。

表 4.4 不同环境下存储系统设计方法的对比

部　　件	工作站类型	单芯片 SoC	基于主板 SoC
处理器	最快速度	更小，可能慢至 1/6 ~ 1/4	更小，可能慢至 1/6 ~ 1/4
缓存	二到三级，非常大(4 ~ 64MB)	简单，单级（256KB）	单级，多元素
存储总线	复杂，慢引脚限制	内部，宽，高带宽	混合
总线控制器	复杂时序和控制	简单，内部	混合
内存	非常大(16GB 以上)限制带宽	限制大小（256MB）	在板上特殊集成
内存访问时间	20 ~ 30ns	3 ~ 5ns	混合

工作站和基于主板的存储设计对于设计者来说是一个很大的挑战。此时需要特别关注缓存，因为它必须能够弥补主存储器位置带来的性能损失。

4.2.3 存储器大小

通过本章内容，不难看出，如何设计大容量片外存储器成为了主板系统设计中的主要问题。由此看来，为什么不限制存储器的大小以期能够合并到片上呢？在虚拟存储系统中，应用程序仍然可以访问大容量的地址空间。对于工作站来说，应用程序需要的环境（以操作系统作为代表）有了很大程度的增长（见图 4.3）。随着环境的继续增长，工作集和活动页也继续增长。这样就需要更多的实际（物理）内存来承载足够多的页来避免过多的页替换操作。过多的页替换操作会损害性能。基于主板的系统面临着稍微不同的问题。在这儿，多媒体数据集通常非常巨大并且需要很大的存储器带宽及多媒体处理器的处理能力。然而，基于主板的系统有一个优势，只要访问带宽足够满足需求，访问时间将不会成为问题。那么能够将多少存储器放到一个芯片上呢？这取决于技术工艺和所需求的性能。表 4.1 也给出了不同技术工艺所需要的面积。eDRAM 通常需要相对大的存储阵列（本章稍后介绍）。因此，例如在一个 45nm 工艺的条件下，预计能有大约 49.2kA/cm^2 或者大约 8MB 的 eDARM。高级电路设计和技术能够显著提高容量，但是大约 64MB 将会是一个极限，除非能出现一种合适的闪存技术。

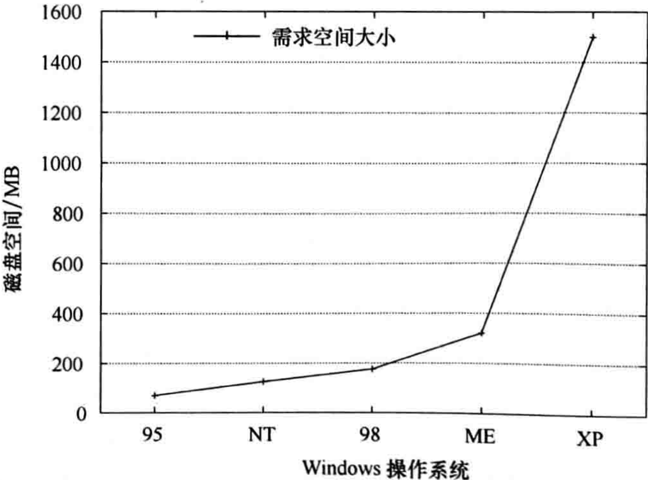


图 4.3 几代微软操作系统需求的磁盘空间（最新的 Vista 系统需要 6GB）

4.3 暂存器和缓存

相对小一点的存储器通常比大一点的存储器访问速度更快，因此将经常访问（或是期望会被经常使用）的指令集和数据存储在一块小的更容易访问（一个周期的访问时间）的内存中会非常有用。如果内存被程序员直接管理，那么这块内存被称为暂存器；如果被硬件管理，那么被称为缓存。

内存管理是一个烦琐的过程，大多数通用计算机仅使用缓存存储器。然而，SoC 提供了可以使用暂存器的可能。假设使用的是众所周知的应用程序，程序员就能够预先明确地控制数据传输。除了缓存控制硬件能够提供出额外的区域来增加暂存器的大小，这样也可以提高性能。

SoC 中实现的暂存器通常用于存储数据而不是指令，因为简单缓存已能处理好指令，效率已经很高。更进一步地说，通过复杂的编程来直接控制指令传输是不值得的。

本章剩下的部分介绍了缓存存储器的理论和设计经验。由于介绍缓存的内容非常多，所以人们很容易遗忘更简单、更早的暂存器方法，但是在 SoC 系统上，有时候这种简单的方法取得的效果却是最好的。

缓存的工作原理是基于程序执行的局部性原理[113]。以下是相关的三个原理：

1. 空间局部性。对于一块特定的内存区域，在程序执行的所有时间内，这块区域或者相邻的区域将会有更高的可能性被访问。

2. 时间局部性。对于对 n 个区域的一系列访问，在一段时间内对于相同区域将会有更高的可能性被访问。

3. 序列性。对于对指定区域 s 的访问，对于接下来的几个访问，很有可能访问的区域是 $s+1$ 区域。这是空间局部性的特例。

缓存设计者们必须一方面满足处理器访问的各种需求，另一方面也要满足内存的各种要求。有效的缓存设计方法会在各种消耗之间取平衡点。

4.4 基础概念

处理器访问的内容存于缓存中时称为缓存命中。访问的内容不存于缓存中时称为缓存失效。在缓存失效的情况下，缓存会从内存中取出失效的内容并放到缓存中。通常情况下，缓存在内存中取出的相关区域的内容称为缓存“行”。行中的内容包括一个或多个物理地址中的字，这些内容是从更高一级的缓存或是主存中取出。物理字是访问内存的基本单位。

处理器-缓存接口定义了一系列的参数。其中直接影响处理器性能（见图 4.4）的参数有以下几种。

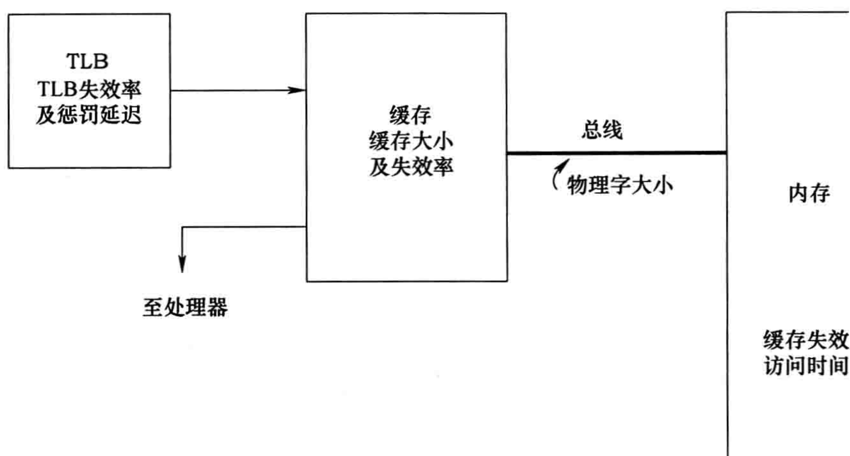


图 4.4 影响处理器性能因素

1. 物理字——处理器和缓存的基本传输单位。

典型的物理字大小：

2～4 字节——最小值，在小核类型处理器中使用。

8 字节及更大——多发射指令处理器（超标量）。

2. 块大小（也称行）——通常是缓存和内存的传输单位。它包括 n 个从内存经由总线传输过来的物理字。

3. 缓存命中访问时间——这个是缓存大小和组织形式固有的属性。

4. 缓存失效访问时间——这个是内存和总线的固有属性。

5. 虚拟地址到物理地址的转换时间（不在 TLB 的转换时间）——地址转换设备的固有属性。

6. 每个周期处理器的访问请求数量。

缓存性能是由失效率或是访问缓存没有命中的可能性计算的。失效时间为失效次数乘以失效后的延时时间。在简单的处理器中，当出现缓存失效后，处理器会暂停执行。

缓存是处理器的一部分吗？

对于许多 IP 设计来说，一级缓存通常是集成到处理器内部的，那么，为什么需要弄清缓存的细节呢？需要弄清缓存的哪些细节呢？最显而易见的回答是 SoC 是由多处理器组成，多处理器之间需要共享内存，通常的共享方式是用一个二级缓存。更好的回答是，一级缓存的细节有可能成为获取存储一致性和合适的程序操作的基础内容。所以，缓存是 SoC 中一个单独的重要的部分。要设计的是 SoC 存储层次，不是孤立的缓存结构。

4.5　缓存组织形式

缓存通常使用按需取或者预取策略。前者的组织结构在简单的处理器上使用广泛。一个命令请求访问策略的缓存只有当失效发生的时候才会在缓存中形成存储局部性。预取策略试图通过局部性原理猜测下一个请求来预取相应的值。它通常在指令缓存中使用。

有三种基本的缓存组织形式：全相联（Fully Associative，FA）映射（见图4.5）、直接映射（见图4.6），以及组相联（见图4.7，实际上是前两者的混合方式）。在一个FA缓存中，当发出一个请求时，地址会和目录中所有的地址条目进行对比（COMP）。如果找到请求的地址（目录命中），缓存相应地址的内容会被取出；否则，缓存失效。

在直接映射缓存中，行地址的低位比特与目录比较（即图4.8所示的索引比特）。由于多个行地址会映射到相同的缓存目录中，所以需要行地址的高位（标签位）与目录地址进行对比来验证是否命中。如果比较结果无效，结果为未命中。直接映射缓存的优势是在访问自身缓存的同时能够同时访问目录表。

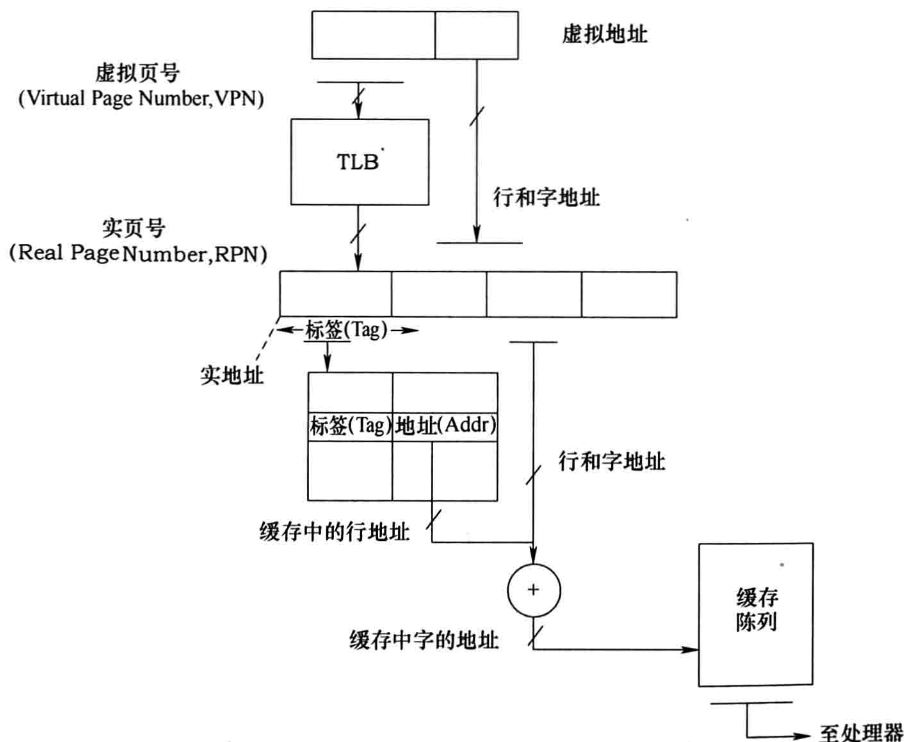


图4.5　全相联映射

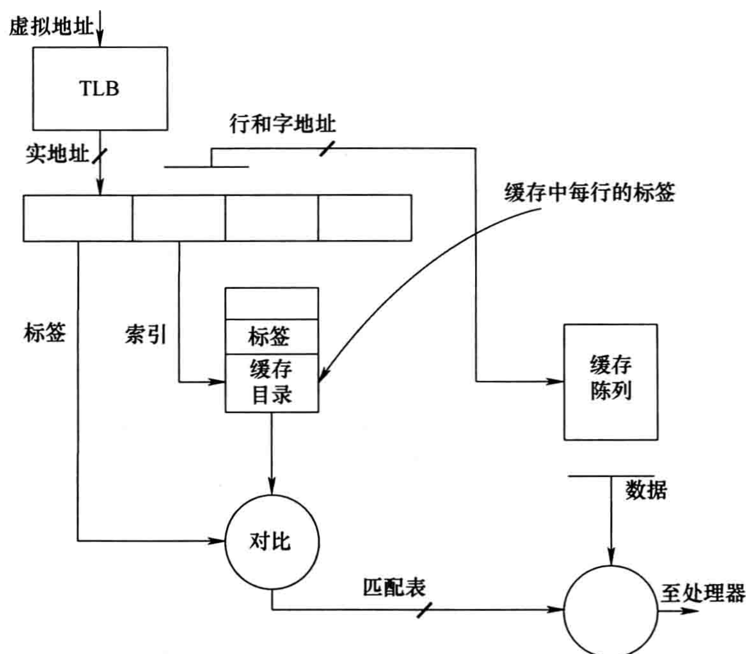


图 4.6 直接相联映射

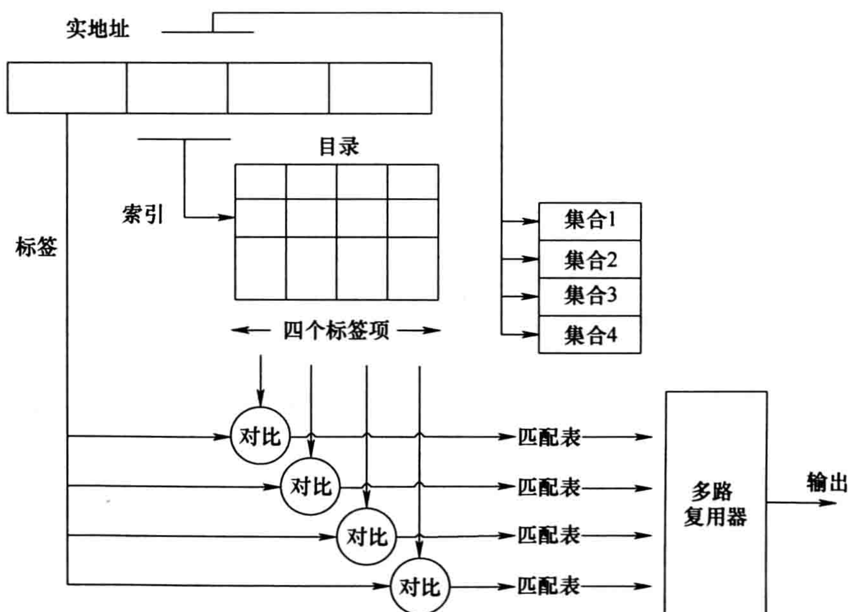


图 4.7 组相联（多直接映射缓存）映射



图4.8　使用时缓存地址的分割

处理器发送给缓存的地址被分为多个部分，每一部分在存取数据的时候具有不同的作用。如图4.8所示的地址分割一样，对比操作（和目录中行地址的高位）中高位区域称为标签（tag）。

下一个部分称为索引（index），用来在缓存目录中和缓存行地址进行对比。标签加上索引就是内存中的行地址。

下一个部分是指定（offset），它是在一行中确定物理字的地址。

最后一部分，是低位区域，能够确定一个字中的字节。这些位在缓存中通常不使用，因为缓存访问通常都是字大小的。当写一个字的某一部分的时候会引发异常。

组相联缓存和直接映射方式类似。行地址的位用来确定缓存目录地址。然而，现在有多种选择，2路、4路或者更多的行地址相联方式。每个地址对应于一个子缓存空间。所有的子缓存空间组成整个缓存阵列。这些子阵列和目录可以被同时访问。如果缓存目录中的任何一个项与访问地址匹配，证明访问命中，缓存子阵列的值被返回给处理器。虽然匹配的过程增加了缓存的访问时间，但是组相联缓存的访问时间优于全相联映射缓存。但是直接相联缓存映射方式提供了最短的处理器缓存访问时间，无论缓存有多大。

4.6　缓存数据

缓存大小在很大程度上决定了缓存的性能（失效率）。缓存越大失效率就会越低。几乎所有的缓存失效率数据都是经验所得，并且有特定的限制。缓存数据具有较强的程序依赖性。同样的，这些数据也常常基于老的机器，在这些机器上，内存和程序大小都是固定的并且很小。这些数据说明了尺寸相对较小的缓存具有较低的失效率。在这种情况下，出现一种倾向，在特定大小的缓存下，测得的失效率随着时间推移有所增加。这些仅是在程序大小逐渐增大的情况下测得的结果。前段时间，Smith[224]发明了一系列的设计目标失效率（Design Target Miss Rate，DTMR），它能够估算出设计者可以从统一缓存（不区分指令和数据）预计得到什么样的结果，如图4.9所示。这些数据是基于缓存功能和行大小得出的典型缓存失效率。

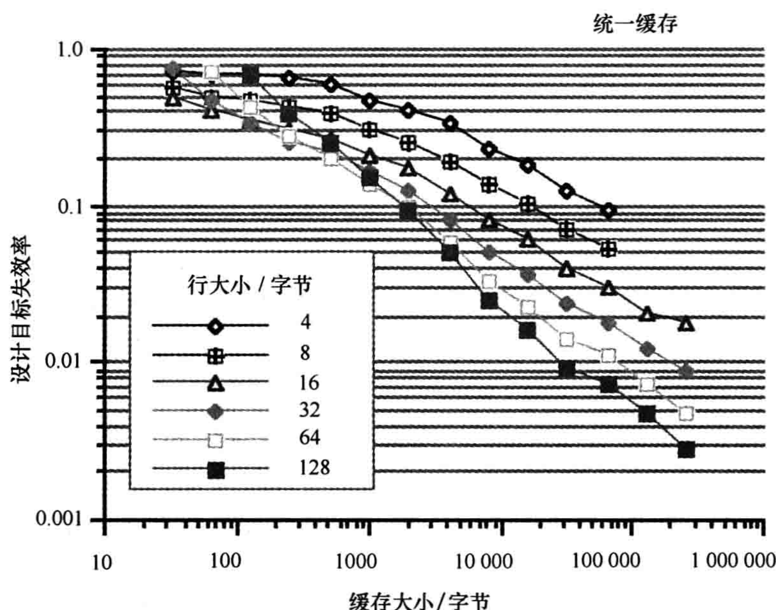


图 4.9 每次访存的设计失效率（全相联，命令取值，分配写取值，LRU 再复制式策略）[223][224]

对于缓存容量大于 1MB，一个常用的规律就是缓存的大小增加一倍失效率就降低一半。这个常用规律对于基于事务的程序来说没那么有效。

4.7 写策略

当发生写操作时，主存是怎样更新的呢？一种方法是同时写缓存和主存（直写[㊀]），另一种方法是仅写缓存（再复制[㊁]，有时候也叫写回），当这一缓存缓存块发生替换的时候写回到主存中。这两种策略是基本缓存写策略（见图 4.10）。

每一次 CPU 存储操作，采用直写策略的缓存（见图 4.10a）的内容既存储在缓存中也存储在主存中。

优点：这让运行的程序在内存中呈现的是一个一致的（最新的）场景。

缺点：内存带宽有可能很高——主要是写数据传输。

在写回式缓存（见图 4.10b）中，当缓存行被替换的时候，新的数据才被写回到内存中。这要求持续记录缓存行的修改（或是“脏”）位，但这样做的优势是减少了内存写操作的数据流：

1. 当缓存任意一行发生写操作，相应的修改位会被置位。

㊀ 直写：Write Through，WT。

㊁ 再复制：Copy Back，CB。写回：Write Back。

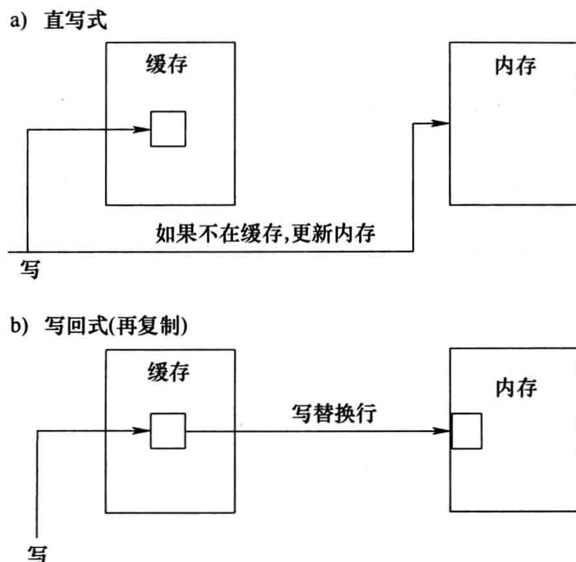


图 4.10　写策略

a) 直写式缓存（写不分配）　b) 写回式缓存（写分配）

2. 从各种各样的记录（Traces）[223] 来看，平均一行的替换率为 47%（20%～80%）。

3. 经验法则：被替换的数据行中有半数是修改的。因而，对于数据缓存，假设有 50% 是修改的；对于统一缓存，假设有 30% 是修改的。

大多数大型缓存使用的是写回策略；直写式缓存仅局限于小型缓存或是特殊用途缓存，这些缓存能够提供及时的内存更新映像。最后，当一次写操作（存储操作）在缓存中失效该如何处理呢？可以从内存中获取这一行的内容（写分配㊀），或者直接写回到内存中（非写分配㊁）。多数直写式缓存采取不分配策略（Write Through No Write Allocate，WTNWA），而大多数写回式 Cache 采取分配策略（Copy Back Write Allocate，CBWA）。

4.8　失效替换策略

缓存失效后会发生什么？如果访问地址没有在目录中被找到，那么一次缓存失效即发生。发生失效情况后，需要迅速采取两步操作：（1）失效的行需要从内存中获取出来；（2）缓存中的某一行需要被当前所访问行（失效行）的内容替换出去。

㊀ 写分配：Write Allocate，WA。

㊁ 非写分配：No Write Allocate，NWA。

4.8.1 读取一行

在直写式缓存中，读取一行的操作包括访问失效行的内容并且将内容替换到缓存行中。

对于写回式策略，需要首先决定要替换掉的行是否是“脏”行（已经被写过）。如果行未被写过，处理方式和直写式的一样。然而，如果行是脏的，必须将脏行写回到内存中。

在访问一行时，最快的访问方式是非阻塞缓存或者预取缓存。这种方法在直写式和写回式缓存中都可适用。在此，缓存拥有额外的控制硬件，可以允许缓存失效被处理（或旁路掉），同时处理器可以继续执行。这种策略仅适用于当访问的数据不是处理器目前所需要的情况下。处理器使用非阻塞式缓存加上编译器提供的预取行机制，可以取得非常好的效果。非阻塞式缓存的有效性依赖以下两方面：

1. 处理器继续执行时，可以旁路掉的失效缓存的数量；

2. 预取的有效性和足够的缓冲区来保存预取的信息；期望使用之前预取的时间越长，失效延迟就会越小，但是这也意味着缓冲区或是寄存器会被占用，因此当前不能被使用。

4.8.2 行替换

当缓存满了的时候，通过替换策略选择一行用来替换掉。有三种替换策略被广泛使用：

1. 最近最少使用（Least Recently Use，LRU），即最近一段时间最少被访问的（读或者写）会被替换掉。

2. 先入先出（First In-First Out，FIFO），即在缓存中最久的会被替换掉。

3. 随机替换（Random Replacement，RAND），即随机选择要替换的行。

因为 LRU 策略符合时间局部性的概念，所以大多数都会选择这个策略。同时这种方法的实现也是最复杂的。每一行都会有一个计数器，读或者写操作的时候就会更新。由于这些计数器可能会很大，所以常通过使用较小的计数器来近似真实的 LRU 策略。

虽然 LRU 的效果比 FIFO 或者 RAND 要好一点，但是使用简单的 RAND 或是 FIFO 相对于 LRU 来说仅增加了 10% 的失效率[223]。

4.8.3 缓存环境：系统、事务和多道程序的影响

许多可用的缓存数据都是基于研究用户程序的记录（trace）得来的。真正的应用程序运行在系统的上下文（context）。基于用户程序运行的经验可知，操作系统的失效率会有轻微的增加（20%左右）[7]。

多道程序环境对缓存提出了特殊的要求。在这种环境下，缓存失效率有可能

不会随着缓存大小的增加而受到影响。环境有以下两种：

1. 多道程序环境。操作系统和一些程序会常驻内存。系统会控制程序在执行完成 Q 条指令之后跳转执行其他的程序，如此继续，直到最后返回到第一个程序。这种环境会形成所谓的“热缓存”的结果。当一个进程返回后继续执行，它会发现一些（不是全部）自己最近使用的缓存行存于缓存中，这样增加了预计的缓存失效率（图 4.11 所示的结果）。

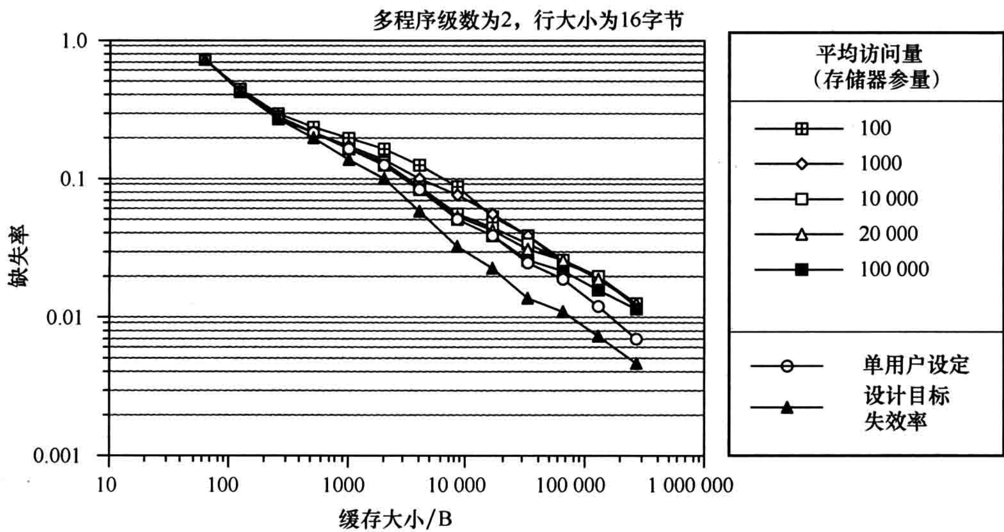


图 4.11　预热缓存（多道程序环境中执行 Q 条指令后任务切换时缓存的失效率）

2. 事务处理环境。当操作系统和一定数量的程序常驻内存时，一些短的应用程序（事务）运行至结束。每个应用程序包括 Q 条指令。这种环境有时候被称为“冷缓存”。(见图 4.12)。

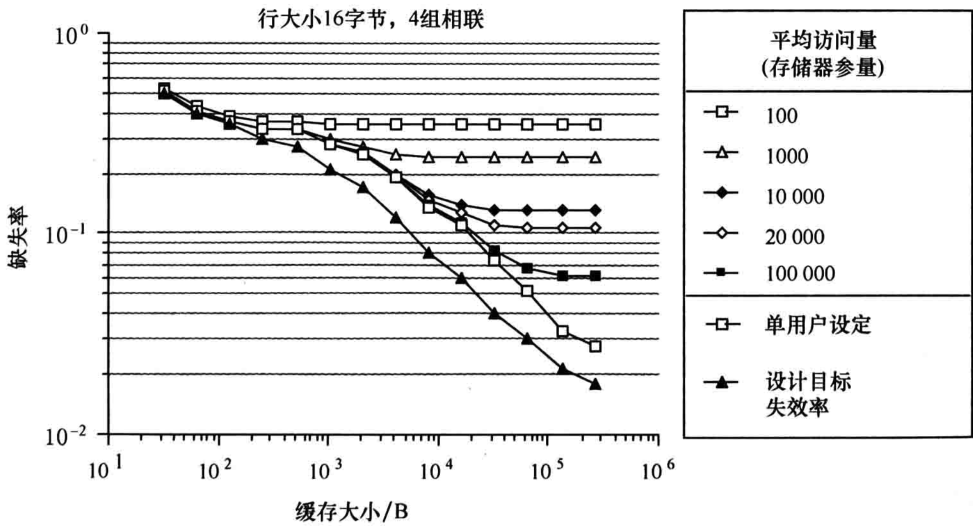


图 4.12　冷缓存（事务处理环境下执行 Q 条指令后任务切换时缓存失效率）

以上描述的这两种环境都有一个明显的特征，在加载完工作集之前，控制权会从一个程序传递给其他程序。这样会大大增加缓存的失效率。

4.9 其他类型的缓存

目前为止，仅考虑了简单的集成缓存（也被称为“统一”缓存），既包括数据缓存也包括指令缓存。接下来将探讨不同种类的其他缓存。表 4.5 给出了缓存的基本类型，但很难将所有的缓存描述详尽，其中包括多种用于特殊甚至是普通应用的缓存设计。

表 4.5 缓存的基本类型

类 型	常 用 位 置
集成(或统一)	基本缓存类型，适合所有访问类型（指令和数据）。常用于二级或更高级缓存
分离缓存(指令和数据)	提供更多的访问带宽，失效率也会上升。常用于第一级处理器缓存
扇形缓存	提高了片上缓存面积有效性（给定区域的失效率）
多级缓存	第一级快速访问；第二级通常比第一级大很多，用来减少一级缓存失效带来的延迟
写集合缓存	特殊化的类型，减少写拥塞，通常使用 WT 片上一级缓存

目前可用的微处理器中，大多数使用分离的指令缓存（I-Cache）和数据缓存（D-Cache），下面会提到。

4.10 分离的指令缓存和数据缓存及代码密度的影响

多缓存结构可以集成到一个处理器设计中，每一个缓存服务一个指定的处理器。多年以来，针对于系统代码和用户代码的特殊缓存，甚至特殊的输入/输出(I/O)缓存都被研究过。最流行的缓存划分配置方式是将指令和数据缓存分开设计。

分离的指令和数据缓存可以带来很大的缓存带宽增长，并且使得整体缓存的访问能力提高了一倍。但是，指令缓存和数据缓存会带来一些消耗；相同大小的统一缓存的失效率会比数据缓存和指令缓存的失效率低一些。在统一缓存中，指令和数据工作集单元的比例随着程序的执行会不断改变，同时也会受到替换策略的影响。

分离的缓存拥有实现上的优势。这些缓存不需要平均分配，因为 75%：25% 的分配比例或是其他比例被证明会更为有效。同样，指令缓存因为不需要考虑存储操作，所以在实现上会相对简单。

4.11 多级缓存

4.11.1 缓存阵列大小的限制

缓存由SRAM存储阵列单元组成。随着阵列大小的增加，需要访问到最远单元的连线也跟着变长。这会最终影响缓存的访问延迟。访问延迟计算公式由缓存大小、组织形式及处理技术组成（特征尺寸f）。McFarland[166]给出了延迟的建模并且找到了近似计算公式如下：

$$\text{Access time} = (0.35 + 3.8f + (0.006 + 0.025f)C) \times \left(1 + 0.3\left(1 + \frac{1}{A}\right)\right)$$

式中，f的单位为μm；C为缓存阵列容量，单位为KB；A为相联度（直接相联为1）；Access Time的单位为ns。

公式如图4.13所示（$A=1$）。如果限制一级缓存的访问时间低于1ns，那么将有可能将缓存阵列的大小限制在32KB。因为可能交叉使用一些不同的阵列，交叉结构也有一定的开销。所以通常情况下，一级缓存低于64KB；二级缓存通常低于512KB（有可能交叉形式使用更小的阵列）；三级缓存使用256KB多种阵列或者更多来形成大的缓存，通常会受到芯片大小限制。

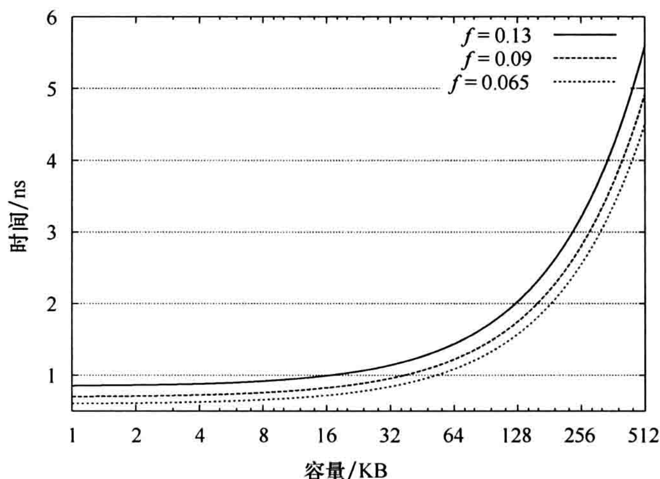


图4.13 缓存访问时间（单阵列为例）与缓存阵列容量函数

4.11.2 评估多级缓存

在多级缓存的情况下，可以使用一级缓存数据来评估这两级缓存性能。一个两级的缓存系统，如果高层级的缓存（一级）所有的内容都包含在低层级的缓存（二级）中，那么称其为“包含缓存”。

二级缓存通过包含原则来获得分析数据。也就是说，一个大的二级缓存包括一级缓存中相同的行。因此，为了评估性能，可以假设一级缓存不存在。决定二级缓存发生的所有缓存失效数量时，可以假设处理器的所有请求直接发送给二级缓存，而没有经过一级缓存。

在二级缓存设计的时候会考虑去适应已经存在的一级缓存。二级缓存的行大小应当和一级缓存的相同或更大。否则，如果二级缓存的行大小小于一级缓存，加载一级缓存中的数据行时会引起二级缓存的两次失效。进一步来说，二级缓存应当远远大于一级缓存，否则，二级缓存将起不到什么作用。

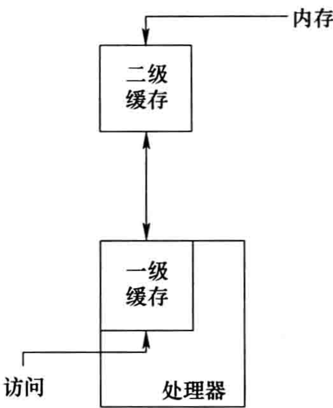


图 4.14　两级缓存示意图

在一个两级缓存系统中，如图 4.14 所示，对于一级缓存和二级缓存，我们将失效率定义如下[202]：

1. 局部失效率就是简单地将失效数量除以访问请求数量。这是通常意义上理解的失效率。

2. 全局失效率是二级缓存失效数量除以处理器的请求数量。这是对二级缓存最基本的测量。

3. 单独失效率是指二级缓存的失效率，如果二级缓存是系统中唯一的缓存的话。这是在包含原则下定义的失效率。如果二级缓存包含所有的一级缓存，可以在忽略一级缓存的情况下计算二级缓存的失效率和处理器的访问率。包含式缓存的原则是基于全局失效率和单独失效率一致的基础，这可以在设计上使用单独失效率来评估全局失效率。

先前的数据（只是读失效率）说明了一些在多级缓存分析和设计时的要点：

1. 只要一级缓存和二级缓存一样大或者更大，在包含的原则下，可以很好地评估二级缓存的行为。

2. 当二级缓存远远大于一级缓存时，二级缓存可以被认为与一级缓存相互独立。它的失效率和单独失效率相当。

例 4.1



失效代价：

一级缓存失效，二级缓存命中：	2 个周期
一级缓存失效，二级缓存失效：	15 个周期

假设我们有一个两级缓存，失效率分别是 4%（一级缓存）和 1%（二级缓存）。假设一级缓存失效、二级缓存命中的开销是 2 个周期，两者都失效的时间开销是 15 个周期（比二级缓存命中多 13 个周期）。如果处理器每条指令访问一次缓存，可以计算出由于缓存失效所引起的额外的 CPI 的值：

一级缓存失效时额外的 CPI 值

= 1.0 refr/inst × 0.04 misses/refr × 2 cycles/miss

= 0.08 CPI

二级缓存失效时额外的 CPI 值

= 1.0 refr/inst × 0.01 misses/refr × 13 cycles/miss

= 0.13 CPI

注释：二级缓存失效时间开销为 13 个周期，不是 15 个周期，因为二级缓存失效已经在一级缓存额外的 CPI 中占用了 2 个周期，即

所有影响 = 额外的一级缓存 CPI + 额外的二级缓存 CPI

= (0.08 + 0.13) CPI

= 0.21 CPI

目前一些 SoC 处理器的缓存配置如表 4.6 所示。

表 4.6 一些 SoC 处理器的缓存配置

SoC	一级缓存	二级缓存
NetSilicon NS9775[185]	8KB I-缓存，4KB D-缓存	—
NXP LH7A404[186]	8KB I-缓存，8KB D-缓存	—
Freescale e600[101]	32KB I-缓存，32KB D-缓存	1MB（带 ECC）
Freescale PowerQUICCIII[102]	32KB I-缓存，32KB D-缓存	256KB（带 ECC）
ARM 1136J（F）-S[24]	64KB I-缓存，64KB D-缓存	最大 512 KB

4.11.3 逻辑包含

真包含或是逻辑包含不应当和统计学上的包含相混淆。一级缓存所有的内容常驻二级缓存，二级缓存常常包含一级缓存的数据内容，并不总是包含。对于逻辑包含，常会有一系列的要求。显而易见，一级缓存必须是 WT 式的；二级缓存并不需要。如果一级缓存是 CB 式的，一级缓存的行写操作不会立即写到二级缓存中，这样一级缓存和二级缓存的内容就会不一致。

当需要逻辑包含方式时，有必要强制一级缓存和二级缓存的内容相一致，并且使用缓存一致性策略。

共享存储多处理器需要整体一致的存储映象，因而逻辑包含在该类系统设计中是一个首要问题。

4.12 虚实转换

TLB 通过将虚拟地址转换为实地址来向缓存提供地址。

图 4.15 给出了一个两路相联 TLB。页地址（虚拟地址的高位）由需要转换的地址位组成。被选择的虚拟地址位用来选中 TLB 中的项。这些位是从虚拟地

址中被选择的（或是通过哈希选中）。这样避免了大量的地址冲突。例如，当地址和数据也有相同的低位时就会发生冲突，如“000”。虚拟地址的大小等于 $\log_2 t$，t 的值是 TLB 中的项数除以相联度。当一个 TLB 项被访问时，会读取一个虚拟地址和实地址映射对。虚拟地址会和虚拟地址标签（索引位之外虚拟地址位）相比较。如果对比相匹配，相应的实地址会使用多路复用方式从 TLB 中输出。

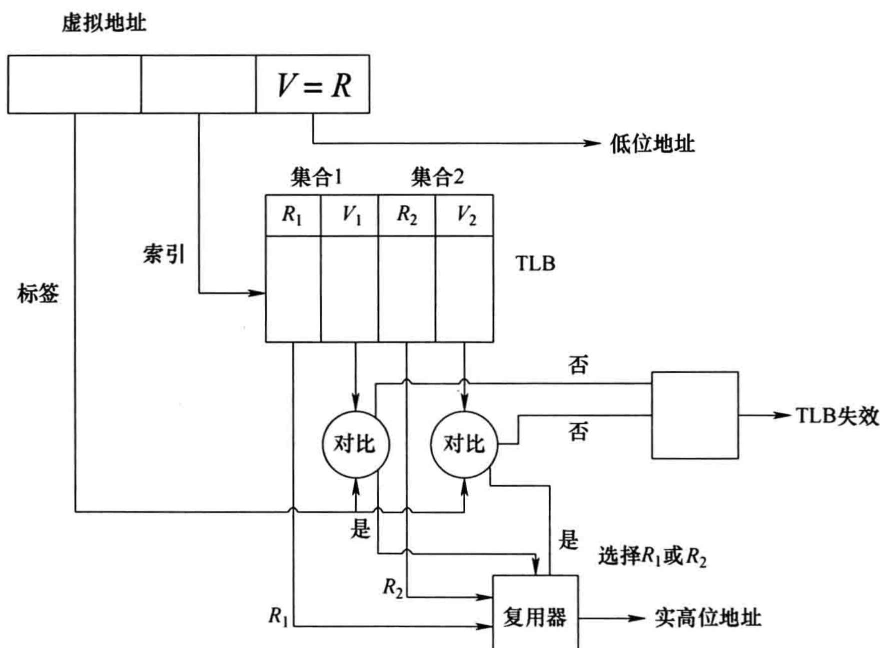


图 4.15 一个两路相联 TLB

尽管页地址的分配非常仔细，但是在访问缓存的同时也会去访问 TLB 的内容。当未在 TLB 中找到转换结果时，需要重复执行操作，在 TLB 中创建一个正确的虚拟-实地址对，操作过程在本书第 1 章有详细描述。这有可能要 10 个周期以上的时间；TLB 失效（没有存于 TLB 中）是非常影响性能的。TLB 访问在许多情况下类似缓存访问。FA TLB 总体来说速度较慢，但是四路或是更高的组相联 TLB 性能较好，总体来说更具吸引力。

图 4.16 给出了典型的 TLB 失效率。FA 数据类似四路组相联的数据。

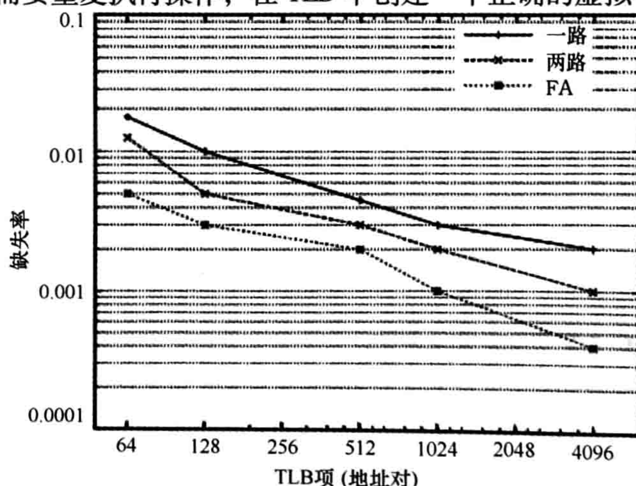


图 4.16 典型的 TLB 失效率

对于那些 SoC 或基于主板的虚拟地址系统，需要额外考虑其他情况：

1. TLB 太小会导致过多的 TLB 失效错误，给程序运行增加了很多时间开销。

2. 如果缓存使用实际地址，访问 TLB 必须在访问缓存之前，这样增加了缓存访问时间。

过多的 TLB 缺失可以通过 TLB 的优化设计来控制。TLB 的大小和组织形式设计需要根据目标系统的性能来确定。

一般来说，大多数的 TLB 使用指令和数据分离转换的方式。这两种 TLB 都可能使用 128 项、两路组相联，以及使用 LRU 替换算法。目前，一些 SoC 处理器的 TLB 配置如表 4.7 所示。

表 4.7　一些 SoC 处理器 TLB 配置

SoC	组织形式	项　数
Freescale e600[101]	独立的-I-TLB，D-TLB	128 项，两路组相联，LRU
NXP LH7A404[186]	独立的-I-TLB，D-TLB	每个 64 项
NetSilicon NS9775 （ARM926EJ-S）[185]	混合	32 项，两路组相联

4.13　片上存储系统

片上存储设计是通用存储系统设计的一个特例，4.14 节将讨论通用存储系统的设计。设计者在存储系统本身的选择及整体缓存-主存组织形式上有更大的灵活性。一旦掌握了应用程序的特性，可以估计出需要存储的程序和数据的大小。很多情况下，部分程序会以固件 ROM 的形式存储。剩余的存储器是由 SRAM 或 DRAM 构成。SRAM 的实现方式是和实现处理器的处理过程一样的，一般情况下，DRAM 则不同。1 个 SRAM 位包括了 6 个晶体管单元，DRAM 则只有 1 个晶体管加上 1 个深沟电容。DRAM 单元设计目标是争取最大的密度，因而使用了很少的连线层。DRAM 的设计目标是低刷新率和加强的低漏电电流。1 个 DRAM 单元使用了纳米级长度的晶体管，更高的临界电压（V_T），就为了能够达到一个更低的漏电电流。这样更容易导致门电路驱动过度及更慢的翻转。对于一颗独立芯片，最终的结果就是 SRAM 的速度比 DRAM 快 10～20 倍，密度是其 1/10 以下。

eDRAM[33,125] 的引入对于片上存储设计来说是一个折中的选择。因为在 SoC 上实现 eDRAM 需要更多的处理步骤，需要大量的加工制造成本，并被看做是硬 IP（或者至少是固 IP）。因为 eDRAM 有一项开销（见图 4.17），所以导致相对于 DRAM 来说密度要小。eDRAM 的设计复杂，包括制造三层额外的掩膜层，这会导致比 DRAM 多 20% 的成本。

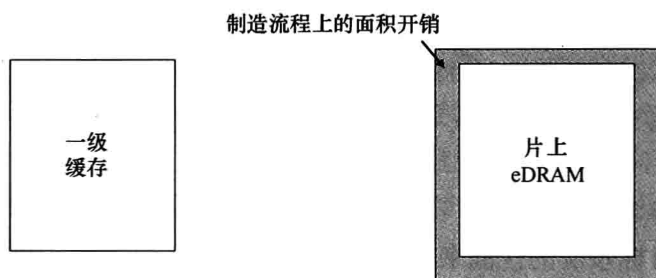


图 4.17 片上 SRAM 和 eDRAM（eDRAM 必须符合处理逻辑的要求，意味着一种开销；SRAM 不受影响）

对于采用 eDRAM 技术的 SoC，可以在同一块芯片上集成高速、高漏电逻辑晶体管和低速、低漏电的存储晶体管。eDRAM 的优势在于集成密度高，如图 4.18 所示。因此选择 eDRAM 而不是 SRAM 的一个重要因素就是基于存储器大小的要求。

eDRAM 在付出加工成本的代价后，其时间参数比传统的 DRAM 好很多。eDRAM 的周期时间（及访问时间）接近于 SRAM，如图 4.19 所示。所有类型的片上存储器都可以借助带宽的优势，每个周期都可以作为一个整体的存储列被访问。

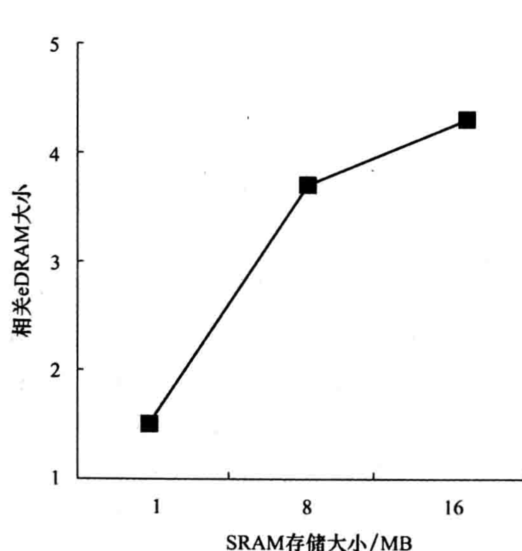


图 4.18 随存储大小改变 eDRAM 的相对密度优势示意图

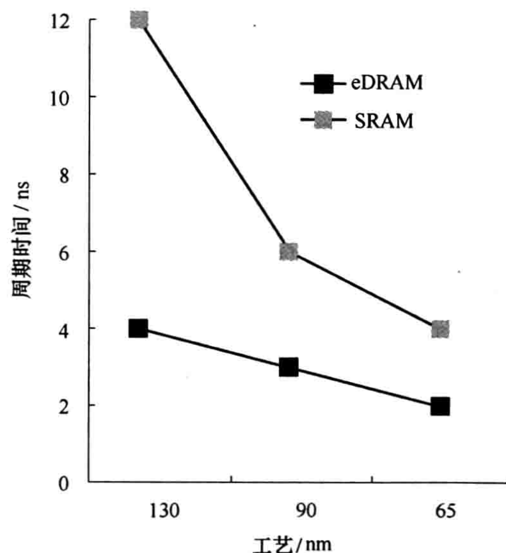


图 4.19 随机访存时间周期数

选择存储器的最后一个因素是考虑到辐射带来的预期错误率（称软错误率[⊖]）。每个 DRAM 单元比 SRAM 单元存储的电荷要多很多。SRAM 单元更快也更容易发生翻转，这样就导致更高的 SER。另外，对于一个 SRAM 单元来说，从工艺角

⊖ 软错误率：Soft Error Rate，SER。

度来看，决定产生一个错误的临界电荷数量会随着增大供给电压和单元容量而降低。不同点如图 4.20 所示。甚至在 130nm 特征大小的情况下，SRAM 的 SER 大约比 eDRAM 高 1800 倍。当然，对于更易产生错误的 SRAM 来说，可以使用更广泛的 ECC 来进行弥补，但这也会带来一些 ECC 的自身的开销。

总而言之，片上存储实现技术的选择依赖制造的复杂程度和存储容量的要求。

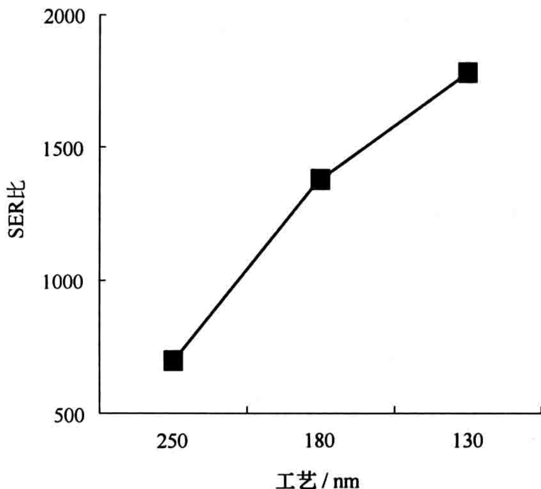


图 4.20　SRAM 相对于 eDRAM 的 SER

4.14　片外（基于主板）存储系统

在众多处理器设计中（可能除了 SoC 外的所有情况），主存系统都是主要的设计挑战。因为处理器的整体设计会相当复杂，所以服务于相应处理器的存储系统的设计也会相应非常复杂。

存储模块包括所有的需要将缓存行通过总线传送给处理器的芯片。缓存行的传送是以总线突发字的传送方式进行的。每个存储模块包含两个重要的参数：模块访问时间和模块周期时间。模块访问时间是指对于一个特定的存储器，当在地址寄存器给出有效的地址时，从存储器中读取一个字到输出缓冲寄存器所消耗的时间。存储器服务（周期）时间是指对同一存储模块的两次直接访问之间的最小时间。不同的工艺导致了访问时间和服务时间之间的巨大差别。访问时间是指处理器访问存储行操作的全部时间。在一个小型简单的存储器系统中，访问时间可能比芯片访问时间加上多路复用和传输延迟稍多一点。服务时间接近芯片的周期时间。在一个大型多模块存储系统中（见图 4.21），访问时间会显著增加，目前来说包括模块访问时间加上传输时间、总线访问开销、错误检测及纠正等延迟。

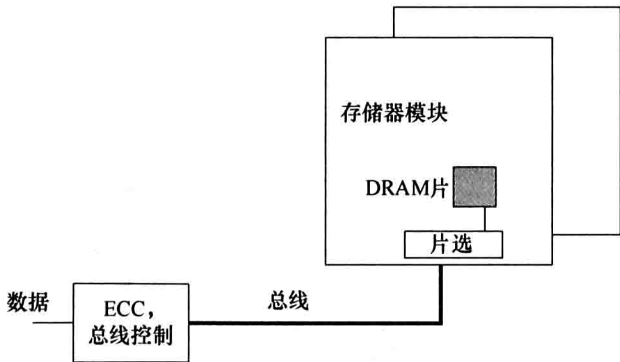


图 4.21　复杂存储系统的访问延迟（访问时间包括芯片访问、组件开销和总线传输）

多年来，随着 DRAM 存取芯片的稳步进展，近几年越来越重视存储芯片性能（而不仅仅是大小）。

对于 DRAM 来说，最重要的成就是同步 DRAM（Synchronous DRAM，SDRAM）的出现。这种技术将 DRAM 访问和周期时间与总线时间同步起来。其他方面的提高加快了总线和模块的数据传输速度，同时提高了它们的电器特性。现在有各种各样的 SDRAM。基本的 DRAM 类型如下：

1. DRAM——异步 DRAM。
2. SDRAM——存储模块时序和存储器总线时钟同步。
3. 双速 SDRAM（Double Data Rate SDRAM，DDR SDRAM）——存储模块在每个总线周期获取双倍长度的传输单元，并且以两倍的总线时钟速率传输。

下面将介绍异步 DRAM 的基础知识。之后将介绍更加高级的 SDRAM。

4.15 简单 DRAM 和存储阵列

最简单的异步 DRAM 包括一个单存储阵列，具有一个（有时候是 4 个或 6 个）输出位。芯片内部是一个行列编址的二维存储单元阵列。如此，内存地址的一半用来识别行地址，$2^{n/2}$ 行中的一行，另一半的地址同样地用来识别 $2^{n/2}$ 列中的一列（见图 4.22）。单元本身保存的数据非常简单，仅包括一个存储电荷的 MOS 晶体管。随着时间的推移，单元一直在放电，所以几毫秒就需要刷新一次。

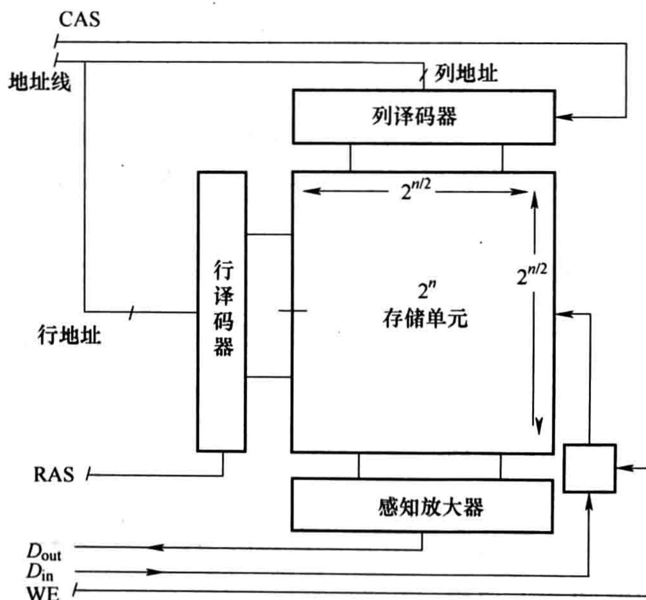


图 4.22 存储芯片示意图

在大容量的存储器中，地址的连线数决定着芯片的引脚数。为了保护这些引脚，也为了获得更好的全局密度而提供一个更小封装，行和列地址复用映射到这些连线（输入引脚），这些同时映射到片内的内容项。有两个额外的连线非常重要：行地址控制器（Row Address Strobe，RAS）和列地址控制器（Colum Address Strobe，CAS）。这些控制门电路首先允许行地址，然后运行列地址进入芯片。行和列地址然后通过译码来选择 $2^{n/2}$ 个可能行的一行。行和列的交叉部分就是需要的信息位。在一次读周期内，列线路信号经过感知放大器放大后传送给输出引脚（数据出口，D_{out}）。在一次写周期内，通过可写（Write Enable WE）信号使数据输入（D_{in}）信号有效，从而用来指定选择的位地址内容。

所有的这些行为都以近似图 4. 23 所示的时序发生。在存储器读操作的一开始，RAS 线就会被激活。当 RAS 激活并且 CAS 关闭的时候，地址线上的信号会被解释为行地址然后保存到行地址寄存器。这样可以激活行解码器并且选择存储阵列里面的行线路。然后 CAS 激活，让列地址线路将信号输入到列地址寄存器中。注意以下一些内容：

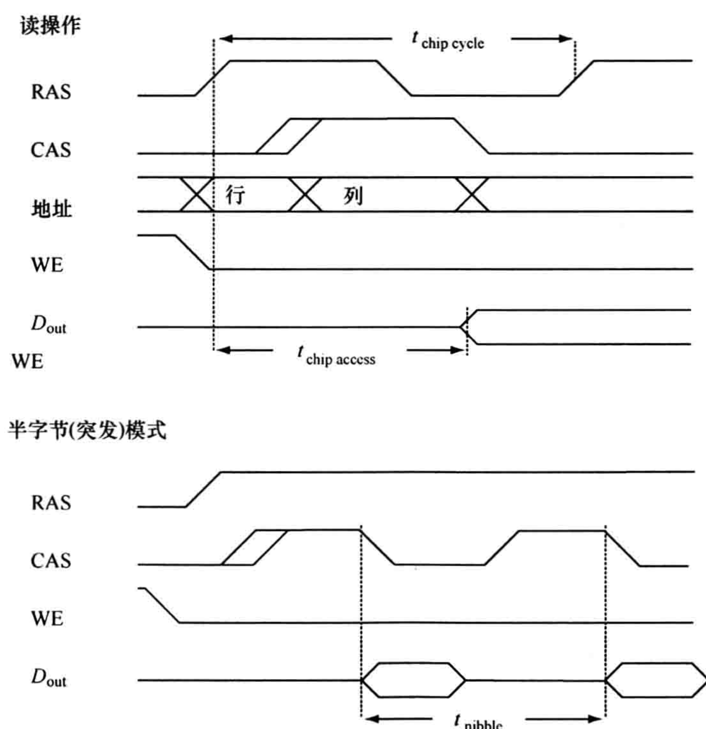


图 4. 23　异步 DRAM 芯片时序

1. 考虑到列地址信号，CAS 的两次上升时序代表着信号的最早和最晚有效时间。

2. WE 信号在读操作的时候一直处于关闭状态。

然后列地址译码器选择一列线路；在行和列的交叉部分就是需要的数据位。在一个读周期内，WE 信号一直处于关闭（低位）状态，输出线路（D_{out}）处于高阻状态直到被低位或是高位信号激活，高位或是低位依赖所有选择的存储单元的内容。

从 RAS 开始有效到数据输出线路被激活，这段时间值对于存储模块的设计非常重要。这段时间被称为芯片访问时间 $t_{chip\ access}$。其他重要的芯片时序参数是存储芯片的周期时间 $t_{chip\ cycle}$。周期时间和访问时间不一样，在此时间内，被选择的行和列线路信号必须在下一次地址能进入及重复读操作之前得到恢复。

异步 DRAM 模块不仅是由一些存储芯片组成（见图 4.24）。在一个存储系统中，假设每个物理字 p 位，一个模块中可以存储 2^n 个字，n 位地址进入到模块并且一般情况下直接进入一个动态存储控制器芯片。这个芯片，与内存时序控制器相连接，提供以下功能：

1. 将 n 个地址位多路复用为行或列地址，给存储芯片使用。
2. 正确的 RAS 和 CAS 信号在合适的时间给出。
3. 提供存储系统的定时刷新。

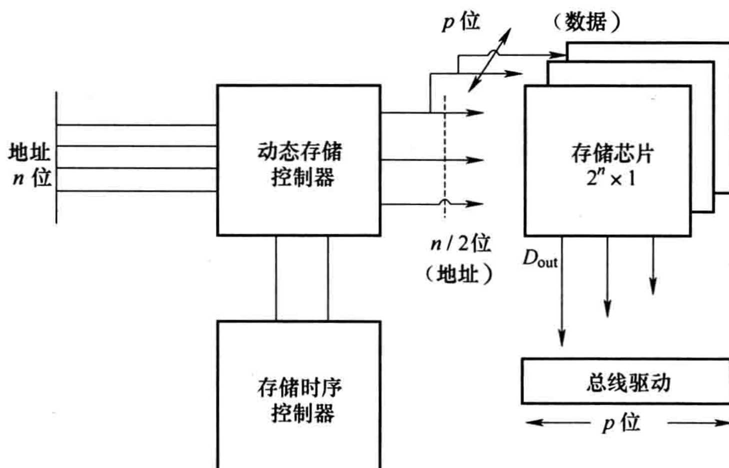


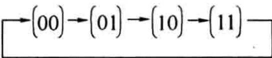
图 4.24 异步 DRAM 存储模块示意图

因为动态存储控制器的输出控制所有物理字的 p 位，也就是 p 个芯片，所以控制器的输出也需要缓冲。当存储器读操作完成后，数据输出信号被连接在存储器的总线驱动，这个总线也是所有存储模块输入/输出的接口。

DRAM 芯片中有两个特点影响存储系统的设计。那些“突发”状态类型特点被称为

1. 段模式
2. 页模式

这两种技术都是为了提高存储字传送速率。在段模式下，单地址（行或列）被提供给存储芯片，CAS 线信号在重复切换。在内部，芯片将这个 CAS 切换转换成列地址低位模数 2^w 的级数。这样，连续的字可以从存储芯片中被高速地访问。例如，当 $w=2$，可以访问 4 个连续的低位地址，如下：



并且返回到原始的位地址。

在页模式下，单独的行会被选择，然后通过重复地激活 CAS 连线（类似段模式，见图 4.23），非连续的列地址会高速地输入进来。通常情况下，这种方式用来填充缓存行。

由于常用术语习惯的不同，此处段模式通常指从一个四字地址边界开始访问（最多）四个连续的字（一个段）。表 4.8 给出了一些 SoC 处理器存储设计，包含了一些 SoC 处理器存储大小、位置和类型。最新的 DDR SDRAM 和之后的一些会在下一部分讨论。

表 4.8　一些 SoC 处理器的存储设计

SoC	存储类型	存储大小	存储位置
Intel PXA27x[132]	SRAM	256KB	片上
Philips Nexperia PNX1700[199]	DDR SDRAM	256MB	片外
Intel IOP333[131]	DDR SDRAM	2GB	片外

4.15.1　SDRAM 和 DDR SDRAM

DRAM 最重要的技术提高是 SDRAM 技术。如前面提到的一样，这种方法，将访问 DRAM 和周期与总线周期相同步。

这种方式有一系列巨大的优势。它消除了分隔存储器时钟芯片生成 RAS 和 CAS 信号的需要。在总线时钟上升沿进行同步。同样的，通过扩展封装方式来适应多输出引脚，有 4、8 和 16 个引脚版本，这样可以允许存储器有更多尺寸的模块。

下面将注意力放在总线和存储总线的接口上，通过利用不同的数据线和地址线，进一步提高了总线带宽。现在，当时钟信号上升时，辅助时钟（complement clock）下降。但是在时钟周期的中间时，时钟信号下降，补充时钟上升。这让同步数据在一个周期内传送两次成为可能：当时钟信号处在上升沿和当辅助时钟处在上升沿时进行传递。通过使用这种方法，可以将总线的数据传输速率提高一倍。采用这种方式的存储芯片就是 DDR SDRAM（见图 4.25）。同样的，当多个列访问同一行时，也可以在选中一行后，让其处于选中状态（激活的），而不是

对于每次内存访问只选择一行和一列，如图 4.26 所示。

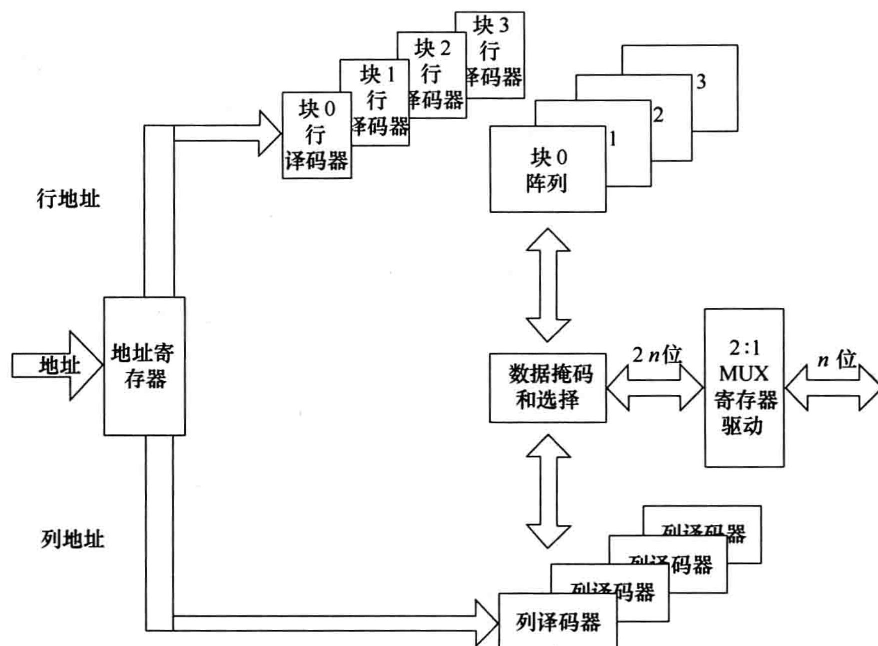


图 4.25 内部 DDR SDRAM 配置

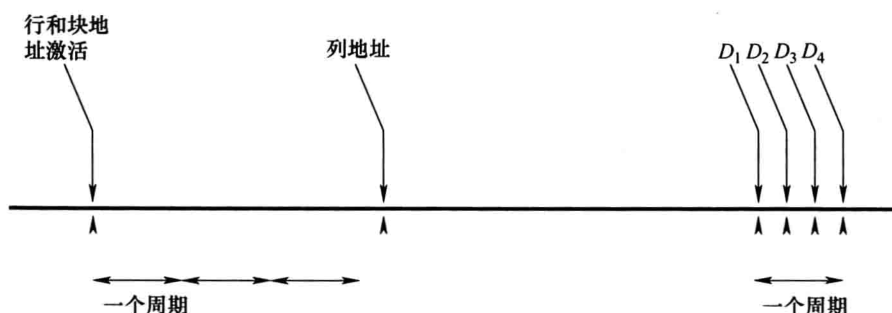


图 4.26 从 DDR SDRAM 读取一行示意图

在空间局部性允许的情况下，通过消除行选择延迟，读和写的次数可以增多。当然，当一个新的行被访问的时候，行激活时间也必须加到访问时间之内。多 DRAM 阵列，通常为 4 或是 8，是 SDRAM 的又一个技术提高。按照芯片的设计方式，用户可以通过程序独立访问或是顺序访问这些阵列。在之前的情况下，每一个阵列可以拥有一个独立的激活行选择，可以提供与多个列交叉访问。如果这些阵列被顺序访问，阵列中相应的行就会被激活，这样可以支持更长的突发连续数据传输。这种方式在图形学的应用中尤其有用。

现代存储芯片时间参数的改善归因于对总线和芯片电气特性的认真关注。除

了使用不同的信号（最初是数据信号，现在是所有信号）外，总线还被设计成带状传输线。随着DDR3（与图形双数据速率非常相关）的出现，终端被集成到片上（而不仅是在总线的尾端），并且还需要特别的校准技术来确保精确终端。

拥有独立阵列支持交叉行访问的DDR芯片必须从这列中获取输出一个$2n$的数据来支持DDR。因此，有四路输出线的芯片必须有可以读取8位的阵列。DDR2阵列典型的特点就是读取$4n$，所以当$n=4$，阵列应当可以获取16位。因为每四路总线半个周期阵列只被访问一次，所以这使更高的数据传输速率成为可能。

表4.9和表4.10给出了一些典型DRAM的参数。在写这本书的时候，这些典型的DRAM已经在生产了，而异步DRAM和SDRAM作为遗留部分通常不会用于新的开发。DDR3部分已经被引入到图形应用的部分。在大多数情况下，这些参数都是典型并针对常用配置的。例如，异步DRAM有1、4、和16输出引脚的配置。DDR SDRAM有4、8和16个输出引脚的配置。也有许多其他的配置种类。

表4.9　64位模块下典型DRAM芯片的配置参数

SoC	DRAM	SDRAM	DDR1	DDR2	GDDR3（DDR3）
典型芯片容量		1Gb	1Gb	1Gb	256Mb
输出数据引脚芯片	1	4	4	4	32
取得阵列数据位	1	4	8	16	32
阵列数	1	4	4	8	4
芯片/模块数	64以上	16	16	16	4
突发传输字	1~4	1，2，4	2，4，8	4，8	4，8，16
行			16K	16K	
列			2048×8	521×16	512×32
32字节行/行/阵列			2048	1024	2048×4

表4.10　一些典型DRAM模块的时序参数（64位）

SoC	DRAM	SDRAM	DDR1	DDR2	DDR3
总线时钟率/MHz	异步	100	133	266	600
启动CAS/ns	30	30	20	15	12
列地址到数据输出1（实际时间）/ns	40	30	20	15	12
行访问（访问新行）/ns	140	90	51	36	28
行访问（一个激活行之内）/ns	120	60	31	21	16
行间隔时间	×1	×4	×4	×8	×1

多种（最多4种）DDR2 SDRAM可以配置使用公共总线（见图4.27）。在这种情况下，当一个芯片被激活后（如有一个激活的行），片上终端就不会被使

用。当芯片上没有被激活的行，终端才被使用。典型的服务器配置，一般有四个模块共享一个总线（通道），一个最多管理两条总线的内存控制器，如图 4.27 所示。之所以数量最多为两个的原因是大量的调谐带和微波传输线必须将控制器和总线连接起来。更高级的技术是在模块和非常高速通道之间增加一个缓冲器。这种高级的通道拥有更小的带宽（如 1 个字节）但是更高的传输速率（如 8 倍速率）。网络的影响使得每个模块的带宽都一样，但是现在接入控制器的线路数量已经减少，却能够使控制器控制更多的通道（如 8 个）。

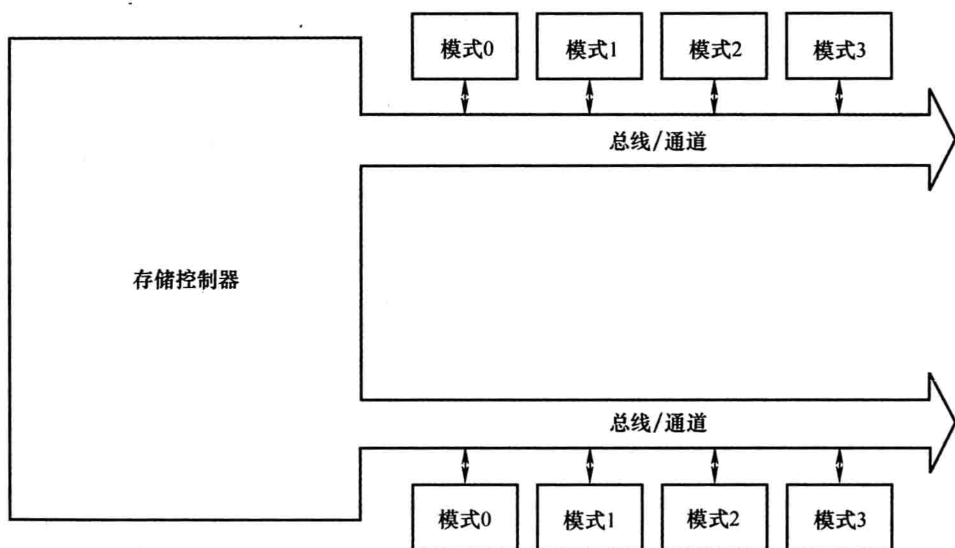


图 4.27　SDRAM 通道和控制器

4.15.2 存储缓冲器

在停止执行和产生更多的访存操作之前，处理器只能保存有限数量的未处理的访存操作。程序中的逻辑依赖或处理器没有足够大空间缓冲未处理的访存操作时就会出现这种情况。这种情况会导致因为处理过程中的暂停，使得可用存储带宽逐渐下降，因为存储器能处理的访存操作和处理器发出请求数量一样多。

逻辑依赖的情况包括分支和地址互锁。在访存操作没有返回结果之前，程序的执行只能挂起。

对于每一个等待处理的访存操作，与之关联是一些特定的信息，用于标注请求属性（如读或者写操作）、内存地址及足够的路由信息用来将请求数据返回给请求者。所有这些信息必须被缓存起来，存储在处理器或者存储系统中，直到访存请求完成。当缓存充满时，后面的请求不会被接受，此时需要处理器暂停。

在交叉存储中，访存模块之间的拥塞情况通常都不相同。多余的访存请求会

发送到相对空闲的模块中，并且会增大网络带宽，因此尽可能增大处理器的请求是非常有用的。如果尽可能增大存储的带宽是主要目的的话，需要存储访存请求达到一定的数量，直到程序中的逻辑依赖成为限制因素。

4.16　处理器-存储器交互简单模型

在一个多处理器或是复杂单核处理器系统中，访存请求有可能阻塞存储系统。要么是多个请求同时产生，导致总线和网络拥塞；要么是不同来源的请求同时访问存储系统。那些不能迅速被处理的访存请求导致对存储器的争用。这种争用竞争降低了带宽，并且有可能使存储器带宽降低。

在这种可能最简单的实现方式下，一个简单处理器发出一个访存请求给一个单独的存储模块。处理器暂停运行并等待存储模块提供服务。当模块有反馈时，处理器恢复运行。在这种实现方式下，结果可以被完全预测。因为在同一时间只有一个访存请求发送给存储模块，所以不会产生竞争。接下来，假设有 n 个简单的处理器访问 m 个独立的模块。当多处理器访问相同的模块时，竞争就会产生。竞争会降低每个处理器获得的带宽大小。更进一步地说，例如，一个配置为非阻塞式缓存的处理器在同一个存储周期产生 n 个请求，这种情况类似 n 个处理器和 m 个模块的存储系统产生的结果，至少从一个建模的角度看是这样的。但是在现代系统中，处理器通常会对存储系统进行缓存。一个处理器是否会在访问缓存的时候因为存储器或者总线竞争而降低速度，取决于缓存设计和分享同一存储系统的处理器的服务率。

假设一个有 m 个模块的集合，每个模块的服务时间为 T_c，访问时间为 T_a，并且给定一个特定的处理器访问率，那么如何模拟从这些存储模块中获得的带宽？如何计算整体有效访问时间？显而易见，那些低优先级插入的模块只能是对带宽大小有帮助的模块，因此它们决定 m 值。从存储系统的角度来看，只要访问的次数统计值保持不变，无论由 n 个处理器组成的处理器系统是否每个存储周期产生一次访存请求，或者是一个处理器每个存储周期产生 n 次访问，对结果的影响都是无关紧要的。因此，对存储系统的分析同样适用于多处理器系统和超标量处理器。访问速率，每 T_c 时间内 n 次访问，被称为发起提交请求率（Offered Request Rate），它代表非缓存处理器系统对主存的峰值需求。

4.16.1　简单多处理器和存储器模型

为了设计一款有用的存储器模型，就需要一个处理器模型。在这里的分析中，用一个单处理器作为一个简单多处理器的模型。对于每个简单处理器来说，当前面的访问被满足后，就可以发起另一次访问请求。在这种模型下，可以改变

处理器的数量和存储模块的数量，并且保持地址访存/数据返回均衡。为了把一个单处理器模型转化为一个对等的多处理器模型，设计者必须决定每个模块服务时间内向存储模块发送的请求数量，$T_s = T_c$。

一个简单的处理器发出一个访存请求并等待回应。一个流水线处理器会对不同的存储器缓冲区发出请求，而不必等待某个访存请求的返回。N 个简单处理器，每个处理器每 T_c 内发出一条访存请求，和一个流水线处理器每 T_s 发出 n 个请求的值是近似等同的（见图 4.28）。

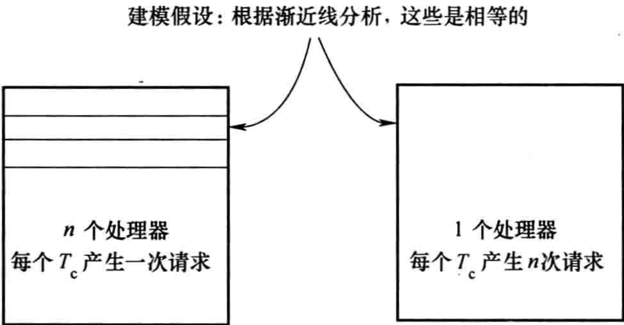


图 4.28 简单的处理器等价寻找方法

在接下来的讨论中，使用两个符号来代表存储系统的带宽值：

1. B。每 T_s 时间内能够响应的请求数。有时候也会给 B 加上参数值，如 $B(m, n)$或者 $B(m)$。

2. Bw。每秒响应的请求数量，Bw = B/T_s。

为了将这些定义转化为基于缓存的系统，响应时间 T_s 是存储系统处理缓存失效的时间。存储模块的数量 m，是存储系统能够一次同时处理的缓存失效的次数。n 是每个 T_s 时间内的所有请求数量。这是每个处理器 T_s 内所有失效次数的预计值乘以所有处理器发出的请求次数。

4.16.2 Strecker-Ravi 模型

这是个简单但有效的竞争评估模型。最初的模型是由 Strecker 搭建出来[229]，之后由 Ravi 独立开发完善[204]。在此模型中，每个存储周期假设有 n 个简单处理器访存请求，且有 m 个存储模块。另外，假设不存在总线竞争。Strecker 模型假设处理器的访存请求模式是统一的，每个请求访问某个特定的存储模块的概率都是 $1/m$。最重要的模型假设是存储系统在每个周期的最开始的状态是不依赖上一个行为的影响的，因此也不依赖过去的竞争。没有响应的请求在存储周期的最后被抛弃。

以下是模型中的近似假设：

1. 只要上一次请求得到满足，处理器就可以发出另一个请求。

2. 每个处理器的访存请求模式都是统一分配的。也就是说，每个请求访问一个特定的存储模块的概率都是 $1/m$。

3. 存储系统在每个存储周期的最初状态是被忽略的，假设在每个周期的最后会丢弃没有响应的请求，处理器此时可以随机地发出新的请求。

分析

用 $B(m, n)$ 代表每个存储周期的平均访存请求，相当于每个存储周期中存储模块的“忙”状态的平均数量。对于给定的模块，每个存储周期产生的事件的角度分析，可以得出：

对于一个处理器来说，Prob（不会访问此存储模块的概率）＝ $(1-1/m)$，

Prob(没有处理器访问这个存储模块的概率) = Prob(模块空闲的概率)

$$= (1-1/m)$$

$$\text{Prob(模块忙的概率)} = 1-(1-1/m)^n$$

$$B(m, n) = \text{忙模块的平均数量} = m(1-(1-1/m)^n)$$

由于竞争的原因，能够获得的存储带宽比理论的最大值要小。忽略掉之前周期的阻塞的情况下，这种分析结果能够得到乐观的带宽值。当然，这里使用的仍然是简单评估模型。

Bhandarkar[41]已经阐述过，$B(m, n)$在 m 和 n 之间是完美对称的。发现了这个事实后，他研究出更为精确的 $B(m, n)$表达式：

$$B(m,n) = K[1-(1-1/K)^l],$$

式中，$K=\max(m, n)$；$l=\min(m, n)$。

可以使用这个表达式描述一类典型的处理器的整体。

例 4.2

假设有一块双处理器的芯片系统，使用共享存储系统。每个处理器芯片有两个核，即一共四个核，共享一个 4MB 二级缓存。每个处理器在一个周期内发出 3 个访存请求，时钟频率是 4GHz。二级缓存中每次访问的失效率为 0.001。存储系统的平均 T_s 为 24ns，包括总线延迟。

可以根据包含原则，忽略掉一级缓存的细节。所以，每个处理器芯片每个周期会产生 6×0.001 存储器访问，或者每个周期 0.012 个访问。因为一个 T_s 时间内有 4×24 个周期，于是每个 T_s 时间内有 $n=1.152$ 个处理器请求。如果设计在每个 T_s 内有 $m=4$ 个请求，性能的计算公式为

$$B(m, n) = B(4, 1.152) = 0.81$$

相关的性能为

$$P_{\text{rel}} = \frac{B}{n} = \frac{0.81}{1.152} = 0.7$$

如此，由于存储系统的影响，处理器只能获得 70% 潜在性能值。为了取得更好的性能，需要一个更大的二级缓存（或三级缓存）或是一个设计更精致的存储系统（$m=8$）。

4.16.3 交叉缓存

交叉缓存的处理方式可以和交叉存储器的方式一样。

例 4.3

一个早期的美国英特尔公司的奔腾处理器有 8 路交叉数据缓存。每个处理器周期发出两个访问请求。缓存和处理器的周期时间是一样的。

对于 Intel 指令集，有

Prob（每条指令的数据访问概率）=0.6.

因为奔腾处理器试图在每个周期处理两条指令，于是有

$$n=1.2,$$
$$m=8.$$

使用 Strecker 的模型，可以得到

$$B(m,\ n)=B(8,\ 1.2)=1.18$$

相应的性能为

$$P_{\text{rel}}=\frac{B}{n}=\frac{1.18}{1.2}=0.98$$

也就是说，由于访存竞争的原因，处理器性能降低了 2%。

4.17 总结

缓存为处理器提供了一个比单纯的存储器访问更加快速的访存方式。因此，缓存成为了现代处理器的重要组成部分。缓存失效率大部分是由缓存的大小决定，但是，任何评估缓存失效率的方法都必须将缓存的组织形式、操作系统、系统环境、I/O 等因素考虑在内。由于缓存的访问时间受到其大小的限制，多级缓存成为了片上处理器设计的共同特征。

片上存储设计看起来是相对容易实现，尤其随着 eDRAM 的出现，但是片外存储设计却是非常困难的问题。首要的客观因素是容量；然而，大的存储容量和引脚的限制意味着比较慢的访存速度。即使芯片访问非常迅速，系统的整体开销，包括总线信号传输、错误检查及地址分配，都会增大延迟。

确实，这些开销延迟的增长相应地降低了机器周期数。面对着上百的访存周期，设计者可以使用大的多级缓存来提供足够的存储带宽来匹配处理器的访问率。

4.18　习题

1. 一个 128KB 的缓存，行大小为 64 位，物理字大小 8 位，4KB 个页，四路组相联。使用 CB 式（写时分配）和 LRU 替换策略。处理器生成 30 位（一个字节的地址）的虚拟地址，转换成 24 位（一个字节的地址）的实字节地址（标识位 $A_0 \sim A_{23}$，从低位到高位）。

（a）哪些地址位不受地址转换的影响（$V = R$）?

（b）哪些地址位用来寻址缓存目录?

（c）哪些地址位用来和缓存目录中的项进行对比?

（d）哪些地址位添加到（b）中的地址位中来寻址缓存阵列?

2. 针对习题 1 画出缓存的层级关系图，详细程度如图 4.5 ~ 图 4.7 所示。

3. 为 DTMR 缓存（CBWA，LRU）设计（字节大小）计算行大小的计算公式：

（a）4KB 缓存

（b）32KB 缓存

（c）256KB 缓存

4. 假设将失效率定为某个值。在这个值上，CB 式缓存（CBWA）与 WT 式缓存（WTNWA）具有相同的访问量。那么称这个值为跨越点。

（a）对于 DTMR 缓存，找到 16B、32B 和 64B 行的跨越点（失效率）。缓存大小为多少时，这些值会一致?

（b）设计行大小对应缓存大小的跨越点。

5. 习题 1 中的缓存在事务环境中使用 16B 大小的行（$Q = 20000$）。

（a）计算有效的失效率。

（b）近似计算，最佳的缓存大小（最小的缓存大小产生的最低可获得失效率）?

6. 在一个两级的缓存系统中，有

- 一级缓存大小为 8KB，四路组相联，行大小为 16B，WT 策略（写时不分配空间）。
- 二级缓存大小为 64KB，直接映射，行大小为 64B，CB，式（写时分配空间）。

假设一级缓存失效，在二级缓存中命中延迟为 3 个周期；一级缓存失效，在二级缓存中也失效的延迟是 10 个周期。处理器访存频率为 1.5 次每条指令。

（a）一级缓存和二级缓存的失效率为多少?

（b）由于缓存失效，CPI 降低多少?

（c）一级缓存中的所有行在二级缓存中都能找到吗？为什么？

7. 一个特定的处理器，有两级缓存。一级缓存是 4KB 直接映射，WTNWA。二级缓存是 8KB 直接映射，CBWA。两者都有 16B 大小的行，使用 LRU 替换策略。

（a）二级缓存能够包含所有的一级缓存行吗？

（b）如果二级缓存是 8KB 的四路组相联（CBWA），二级缓存包含所有的一级缓存行吗？

（c）如果一级缓存是四路组相联映射（CBWA）并且二级缓存是直接相连，二级缓存包含所有的一级缓存行吗？

8. 假设有如下的参数设置，一级缓存大小为 4KB，二级缓存大小为 64KB。

缓存失效率为

4KB	每次访问 0.10 次失效
64KB	每次访问 0.02 次失效
1 次访问/指令	
3 周期	一级缓存失效，二级缓存命中
10 周期	一级缓存失效和二级缓存失效总时间

由于缓存失效而导致的额外的 CPI 多少？

9. 一个特殊的处理器使用 32 位虚拟地址。地址空间为分段和分页，每个段最大为 1MB，每个页为 512B。在缓存中读写交换的物理字为 4B。

使用 TLB，组织形式为组相联，128 × 2。如果地址位标识位 $V_0 \sim V_{31}$ 为虚拟地址，$R_0 \sim R_{31}$ 为实地址，从低地址到高地址。

（a）哪些位不受地址转换的影响（如 $V_i = R_i$）？

（b）如果 TLB 使用地址的低位部分进行转换（非哈希），哪些地址位用来寻址 TLB？

（c）哪些虚拟位用来对比 TLB 中的虚拟项来决定一个 TLB 命中是否存在？

（d）在最小的情况下，哪些实地址位是由 TLB 提供？

10. 对于一个 16KB 的集成一级缓存（直接映射，16B 行大小）及一个 128KB 的集成二级缓存（两路组相联，16B 行大小），找出二级缓存的单独和局部失效率。

11. 一个特殊的芯片有 16KB 大小的指令缓存和 16KB 大小的数据缓存，都是直接映射。处理器有 32 位虚拟地址，实地址为 26 位，使用 4KB 大小页。指令访问频率为 1.0 次每条指令；数据访问为 0.5 次每条指令。缓存失效延迟为 10 个周期，加上 1 个周期的 4B 行转换延迟。整个行写到缓存之前，处理器处于

阻塞状态。D 缓存是 CBWA 的；使用脏行率为 $w = 0.5$。对于两个缓存来说，行大小为 64B。

（a）由于指令缓存失效造成的 CPI 降低和由于数据缓存失效造成的 CPI 降低为多少？

（b）对于 64B 大小行来说，找出指令缓存和数据缓存目录字节大小和相应的映射区域。

12. 找出两个最新的 DDR3 设备的例子，针对这些设备，更新表 4.9 和表 4.10 所示的数据。

13. 列出 TLB 失效之后必须执行的所有的操作信号。设计者该如何降低 TLB 失效造成的延迟呢？

14. 在例 4.2 中，假设需要相对性能为 0.8。通过 $m = 8$ 时交叉操作可以获得吗？

15. 更新表 4.3 所示的基于 NAND 闪存的时序参数。

16. 对目前的商用闪存（NAND 和 NOR）和当前 eDRAM 的特点做比较。

第5章 互　　联

5.1 引言

SoC 设计一般包括 IP 核的综合，其中每一个 IP 核都是被独立设计和验证的。系统综合人员通过最大限度地估计设计可重用性，来减少开销并降低风险。SoC 综合人员经常遇到的问题是 IP 核的互联方法。

可供选择 SoC 互联方式远远多于传统计算机的总线。首先给出 SoC 互联结构的概述：总线和片上网络（Network on Chip，NoC）。为 SoC 专门设计的总线结构将在下面介绍和比较。相对于基于总线的互联网络，基于交换的互联网络具有更多的选择性。这里将不会考虑点对点的或者全部自制的交换互联网络，这些互联网络不适用于多样的 IP 核。在 SoC 互联中使用的基于交换的互联网络被称作 NoC 技术。

一个 NoC 通常包括一个表层的抽象描述，向设计者隐藏了底层的物理互联。那么依照当前 SoC 的惯例，将互联看做是总线或者基于交换的 NoC。在 NoC 中，交换可以被看成是一个交叉开关，一个直接连接的互联，或者一个多级交换网络。

有大量的关于总线和计算机互联的文献。被连接在一起的单元有时被称作代理（在总线中）或者节点（在一般的互联文献中）；本书简单地使用“单元”一词。由于当前 SoC 互联通常包含适当数量的单元，本章给出了互联选择的一个简化表示。对于片上通信结构更深入的解释可以参见本书参考文献［191］。对于计算机互联网络的一般性讨论，请看本书参考文献[60，77]。

5.2 概述：互联结构

图 5.1 给出了在系统环境下一个 SoC 模块的系统框图。这个典型的 SoC 模块包含了一些 IP 核、一个或者多个处理器。另外，各种类型的片上内存充当了高速缓存，数据存储和指令存储。其他集成在 SoC 中的 IP 核提供处理特定应用的功能，如图形处理器、视频处理器和网络控制单元。

什么是 NoC？

按照 SoC 技术的发展，看起来仅有两种互联的策略：总线和 NoC。那么，什

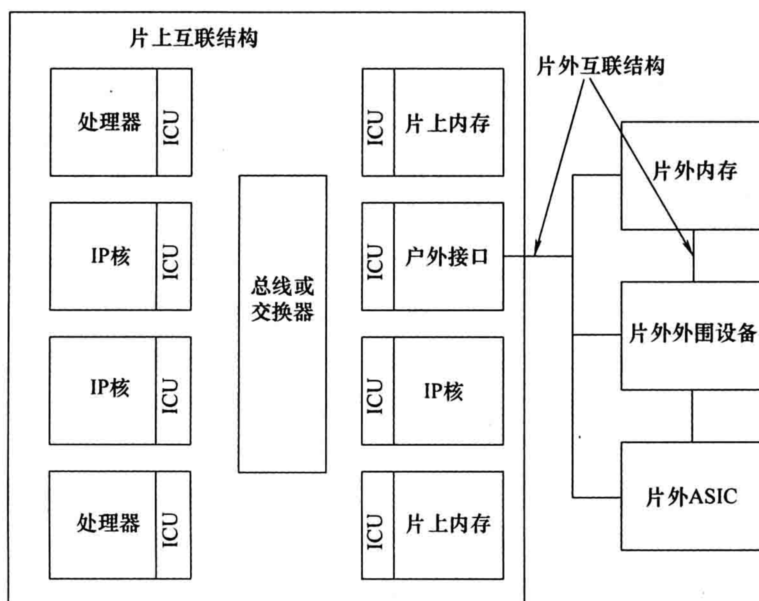


图5.1 在系统环境下一个SoC模块框图

么是真正的NoC? Nurmi教授(在Leibson[167]的报告中提到)总结了NoC的以下特点:

1. NoC不仅是一个单一的共享的总线。

2. NoC为任何通过纵横开关或者节点交换连接在网络中的两个主单元提供点对点的连接。

3. NoC通过并行连接提供高聚集带宽。

4. 在NoC中,通信和计算是分开的。

5. NoC使用层次化的通信方式,即使它由于复杂和昂贵而只有少量的网络层次。

6. NoC支持管道通信并为发送端和接收端之间的立即数据提供缓冲。

在SoC的环境下,设计者发现总线技术不能提供足够的带宽和连接性,互联的选择明显是一些交换开关。任何被很好设计的交换互联可以满足特点2、3、4和6。点对点的交换互联不满足特点5,它的处理器节点和交换互联被特别设计的接口所连接。但是在SoC中,各种IP供应商的点对点互联几乎不是这样的。设计者会选择一个与处理器节点分离的常用的通信接口(层)。

SoC模块中的IP核之间需要互相通信。它们通过互联接口单元(Interconnect interface Unit, ICU)来实现通信。ICU为所有的SoC模块使用一个共同的接口协议。

SoC 模块的外部是片外内存、片外外围设备和大量存储设备。因此，系统的开销和性能取决于片内和片外的互联结构。

选择一个合适的互联结构需要理解以下一系列的系统级问题和说明：

1. 通信带宽，信息在一个模块和它所操作的周围环境之间的传播速率。带宽用 B/s 来衡量，通常一个模块的带宽需求需要扩大互联的类型，从而实现系统的总吞吐量的规格。

2. 通信延迟，一个模块从需要数据到接收到数据反馈的时间延迟。延迟对整个系统的性能来说，也许不是很重要。例如，在视频流应用中的长延迟通常对观看者的体验没有或者有少量的影响。看一部电影时，会比播放延迟几秒，但这没有影响。相反，在移动通信协议中，即使小的，不曾预料的延迟也会造成交流的困难。

3. 主设备和从设备。这两个词语关心的是一个单元是否可以初始化或对通信请求作出反应。一个主设备控制自己和其他模块之间的事务，如处理器。一个从设备，对来自主设备的请求作出回应，如内存单元。一个 SoC 的设计通常包含几个主设备和大量的从设备。

4. 并行需求。独立的同时发生的通信通道的数量。通常，额外的通道可以提高系统的带宽。

5. 信息包和总线事务，在单一事务中的信息传播大小和定义。对于总线来说，由控制位（读/写等）的地址和数据组成。在 NoC 中，这个信息被称作信息包。信息包由头部（地址和控制）和数据（有时被称作负载）组成。

6. ICU。在一个互联中，这个单元管理互联的协议和物理事务。它可以是简单的，也可以是复杂的。它能够支持乱序的事务缓冲和管理。如果 IP 核需要一个传输协议去访问总线，这个单元被称作总线封装。在 NoC 中，这个单元管理信息包从 IP 核传输到交换网络的协议。它提供包缓冲和乱序事务传送。

7. 多时钟域。不同的 IP 模块可能在不同的时钟与数据速率下运转。例如，一个视频摄像机捕获像素数据的速率由视频标准决定，而它的处理器的时钟速率通常由工艺和结构设计决定。因此，在 SoC 中的 IP 核常常需要在不同的时钟频率下工作，从而产生了分离的时间区域，称作时钟域。如果设计不小心，跨时钟域会造成死锁和同步问题。

给定一系列通信规则，设计者可以探索不同互联结构中带宽、延时、并行性和时钟域需求的不同，如总线和 NoC。表 5.1 给出了一些互联结构的实例。其他实例还包括 Altera FPGA 中的 Avalon 总线[10]，用于开源核心和平台的 Wishbone 互联[189]，以及 FPGA 应用中的 AXI4-Stream 接口协议[73]。

为 SoC 设计互联结构需要仔细考虑许多需求，如上面列出的内容。本章的剩余内容给出了两种互联结构的介绍：总线和 NoC。

表 5.1 一些互联结构的实例 [167]

技 术	AMB	AXI(AMBA 3)	CoreCon	Smart	Nexus
公司	英国 ARM 公司	英国 ARM 公司	美国 IBM 公司	美国 Sonics 公司	美国 Fulcrum 公司
核心类型	软/硬	软/硬	软	软	硬
体系结构	总线	单向的	总线	总线	NoC 直接应用
总线位宽	8 ~ 1024	8 ~ 1024	32/64/128	16	8 ~ 128
频率	200MHz	400MHz *	100 ~ 400 MHz	300MHz	1GHz
最大带宽/(GB/s)	3	6.4 *	2.5 ~ 24	4.8	72
最小延迟/ns	5	2.5 *	15	不适用	2

* 在 ARM PL330 的高速控制器中实现。

5.3 总线：基本结构

计算机系统的性能很大程度上依赖它的互联结构的特点。一个不好的系统总线设计可以截断指令和数据在内存和处理器之间的传输，或者在外围设备和内存之间的传输。这个通信瓶颈被许多微处理器和系统制造商关注，在过去的 30 年里，它们采用了许多总线标准。这些标准包括最流行的 VME 总线和 Intel Multibus-II。对于板上系统和个人计算机，总线的发展包括指令集结构总线（Instruction Set Architecture，ISA），EISA 总线和现在的流行的 PCI 和 PCI Express 总线。所有这些总线标准被设计用于在印制电路板（Printed Circuit Board，PCB）上或者在板上系统的 PCB 上连接集成电路（Integrate Circuit，IC）。

虽然这些总线标准用于计算通信很好，它们对于 SoC 技术却不是非常适用。例如，所有这些系统级总线被设计成驱动一个底板，或者在机架系统或者在一个计算机的主板上。这就在总线结构上强加了很多限制。首先，可用信号的数量通常被 IC 上有限的引脚和 PCB 连接器上引脚的数量所限制。向一个 IC 包中添加额外的引脚或者连接器开销昂贵。再者，总线的工作速度常被总线信号的高负载容量、连接器的接触电阻和快速交换信号到达 PCB 所带来的电磁干扰所限制。最后，片上总线的驱动需要更小，以节约面积和功耗。

在详细描述总线操作和总线结构之前，表 5.2 给出了两种互联结构的比较，显示了在一个典型总线从设备上的大小和速度。

表 5.2 两种互联结构的比较[194]

标 准	速率/MHz	面积/rbe *
AMBA（依据实现方式）	166 ~ 400	175 000
CoreConnect	66、133、183	160 000

* rbe 为寄存器等效位；评估是大概的，不同的实现结果不同。

5.3.1 仲裁和协议

概念上，总线仅是被多个单元共享的线路。实际上，必须提供一些逻辑以使总线可以被有秩序地使用；否则，两个单元可能同时发送信号，从而造成冲突。当一个单元独占总线时，这个单元称为拥有这个总线。单元可以是潜在的主单元，它可以要求对总线的拥有权；也可以是从单元，它是被动的，仅可以对请求作出回应。一个总线主设备是这样一种单元，它初始化计算机总线的通信或者输入/输出（I/O）通路。在 SoC 中，一个总线主设备是一个片上组件，如处理器。其他单元连接到片上总线，如 I/O 设备和内存组件，它们被称作“从设备”。总线主设备利用规定的从设备地址控制总线通路和信号。再者，总线主设备也控制主设备和从设备之间的数据流信号。

一个叫做仲裁的部件决定了总线的归属。一个简单的实现是集中式的仲裁单元，它接收来自每一个潜在的请求单元的输入。通过总线协议，仲裁单元将总线占有权授予其中一个请求单元。

总线协议是一系列经过协定的用于在两个或多个设备通过总线传输信息的规则。协议决定了下面的内容：

- 待发送数据的类型和顺序；
- 发送设备是如何知道它已经完成了信息的发送；
- 数据的压缩方式，如果有的话；
- 接收设备如何反馈来表明已经成功接收到信息；
- 仲裁如何解决总线的竞争，按什么优先级解决，以及错误检测的类型使用。

5.3.2 总线桥

总线桥是连接两个总线的模块，这两个总线可以是不同类型的。一个典型的总线桥有如下三个功能：

1. 如果两个总线使用不同的协议，总线桥提供必需的格式和标准的交换。

2. 总线桥在两个总线之间插入，从而分割两个总线，保证信息运输在总线片段中进行。这样就提高了并发性：两个总线可以同时工作。

3. 总线桥常包含内存缓冲区和相关联的控制电路，从而允许写置入。当一个总线上的主设备初始化数据并通过总线桥向另一个总线上的从设备传输数据时，这个数据会暂存在缓冲区；从而允许数据还未真正写入到从设备的情况下，主设备能够继续执行下一个事务。通过允许事务的快速完成，总线桥能够显著提高系统的性能。

5.3.3 物理总线结构

总线事务的特点取决于物理总线的结构（线路通路的数量、时钟周期等）

和协议（特别是仲裁的支持）。在任何给定的周期内，多个总线的使用者必须通过仲裁来访问总线。因此，仲裁是总线事务的一部分。简单仲裁有一个请求周期，在这个周期中，来自不同使用者的信号将被划分优先级别；紧接着是一个反馈接收周期，在这个周期中使用者将被选择。更复杂的仲裁器加入了总线控制线和相应的逻辑，从而每一个使用者会知道即将到来的总线状态和优先级。在这种设计中，没有用于仲裁的周期被加入到总线事务中。

5.3.4 总线多样性

总线可以是统一的，也可以是分离的（地址和数据）。在统一的总线中，地址首先在一个周期中被传送，紧接着是一个或多个数据周期；分离总线有分开的总线完成这些功能。

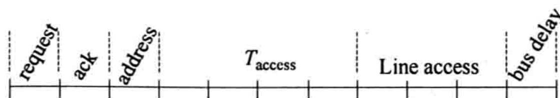
另外，总线可能是单一事务的或者是被占用的。被占用的总线仅在相应的地址或数据周期被事务占据。这种总线有一个单元接收器，可以缓冲信息并产生分离的地址和数据事务。

例 5.1 总线实例

通过组合不同的物理总线宽度和仲裁协议，可能产生许多总线设计方法。下面列出的实例具有显著的可能性。假设总线有 1 个处理器周期的传输延迟，内存（或者共享缓存）在初始化地址后有 4 个周期的访存延迟，并且每一次连续的数据访存都需要额外的周期。每一次能够访问内存 4 个字节。待传输的数据由 16 位的缓存行组成。地址为 4 个字节。

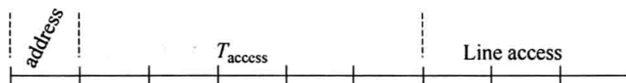
其中，T_{access}表示在地址流出之后访问内存的第一个字所需的时间，$T_{\text{line access}}$表示访问剩余字所需的时间。另外，最后一个字节的数据在这个时间模板的末尾到达，并且它只能在这个时间点后被使用。

（a）简单总线。这是一个单一事务总线，具有简单的请求/反馈收到（ack）仲裁。它有 4 个字节的物理位宽。请求和反馈信号是分离的，但是呈现为总线事务的两个部分，因此总线事务的延迟是 11 个周期。第一个字在 T_{access} 的最后一个周期从内存发送，而第四个（和最后一个）字在 $T_{\text{line access}}$ 的最后一个周期从内存发送。最后一个总线周期用于重置仲裁器。



（b）有仲裁支持的总线。这个总线有一个更加复杂的仲裁器，但是仍旧有 4 个字节的物理位宽和集成的地址和数据。对于将地址从总线缓冲器发送到内存，这个总线有一个额外的访问周期（5 个周期替代 4 个）。这没有在实例（a）中显示出来，因为简单的总线常常比较慢，就直接将地址送入到内存中了。现在请

求和反馈的初始化在总线处理中是重叠的，并且最后用于重置仲裁器的周期没有在本例的图中出现，因此，总线事务现在开销 10 个周期。



（c）独占的分离总线，4 字节位宽。有如下假设：数据从请求线到缓冲器需要 5 个周期，然后传输需要 4 个周期。事务延迟同实例（b）一样为 10 个周期，其中包含了地址传输的周期，事务占据总线的时间少于延迟的一半（4 个周期）。地址总线只使用了 10 个周期中的 1 个。剩余的时间被用于其他不相关的事务，以提高通信的性能。



（d）独占的分离 ，16 个字节位宽，1 个周期总线事务时间。在实例（c）中，事务的延迟为 10 个周期。因为内存显然限制着系统，在这个实例中，在一个周期内传输数据之前，内存获取整个 16 个字节的缓存行。地址总线和数据总线在一个事务中仅被使用 1 个周期。需要指出的是，实例（d）允许一个额外的周期来重新访问总线，尽管这可能不是必须的，并且在实例（c）中没有出现。



实例（c）和（d）很有趣，因为总线的位宽超过了内存的位宽；例如在实例（d）中，内存在 7 个周期中是繁忙的（4 个周期用于访问第一个字，3 个周期用于访问剩余的字），但是总线只在 1 个周期中是繁忙的。在这两个实例中，总线-内存的这种状况是由内存的限制造成的，这就是竞争产生的地方。

5.4 SoC 总线标准

两种常用的 SoC 总线标准是 ARM 开发的高级微控制总线结构（Advanced Microcontroller Bus Architecture，AMBA）和美国 IBM 公司开发的 CoreConnect 总线。后者在美国 Xilinx 公司的 Virtex 平台的 FPGA 系列中采用。

5.4.1 AMBA 总线

AMBA 总线在 1997 年面世，最初源自 ARM 处理器，它是工业界 SoC 处理器中最为成功的一个。AMBA 总线基于传统的总线结构，包含两个层次。在 AMBA

总线规定[22]中有以下两种总线定义：

- 高级高效总线（Advanced High · performance Bus，AHB）被设计用于连接嵌入式的处理器，如 ARM 的处理器核、高性能外围设备、直接内存存取（Direct Memory Access，DMA）控制器、片上内存和接口。它是一个高速的高带宽总线结构，使用分离的地址，读和写总线。标准建议数据操作最小为 32 字节，数据位宽被扩展到 1024 位。支持并发的多主/从设备操作。它也支持数据的突发传输模式和分离的事务。所有在 AHB 上的事务都被关联在一个单一的时钟沿，这使得系统级设计更容易理解。

- 高级外围总线（Advanced Peripheral Bus，APB）比 AHB 的性能低一些，但是能够使功耗降低到最优水平，并且减小了接口的复杂程度。它被用于设计接口，以降低外围模块的运行速度。

第三种总线，叫做高级系统总线（Advanced System Bus，ASB）。它是 AHB 的前身，被用于低性能的系统设计，使用 16/32 位的微控制器。当 AHB 开销、性能和复杂度不合适时，就会用到 ASB。

AMBA 总线被设计用于在 SoC 综合时，产生一系列信息的地址。这些信息来自于 ARM 处理器的使用者。它设计的目标有以下几个：

1. 模块化设计和设计的重用。因为 ARM 处理器的总线接口是非常灵活的，通过使用特别的总线和控制逻辑，经验不丰富的设计者可能会不经意地开发出低效的甚至不可工作的产品。AMBA 总线规格鼓励模块化的设计方法，这可以更好地支持模块化的设计和设计的重用。

2. 定义良好的接口协议、时钟和重置。AMBA 总线定义了一个低开销的总线接口和简单灵活的时钟结构，AMBA 总线的性能被它的多主设备、分离的事务和突发模式操作所提高。

3. 低功耗支持。与其他嵌入式处理器核相比，一个 ARM 处理器吸引人的地方是它的功耗效率。AMBA 总线的两层设计使得它在外围模块的能源效率设计上可以很好地适应低功耗的 CPU 核。

4. 片上访问测试。AMBA 总线有一个可选的片上访问测试的功能，它可以重新利用基本总线的基础设备进行连接到总线上的模块的测试。

AHB 图 5.2 给出了一个典型的 AMBA 总线系统。AHB 是这个系统的主干，ARM 处理器、高位宽内存接口、RAM 和 DMA 设备在主干的两侧。AHB 和慢速的 APB 通过一个总线桥模块连接。

AMBA AHB 协议被设计用于实现一个多主设备系统。不像大多数的为基于 PCB 的系统设计的总线结构，AMBA AHB 通过采用一个中心多路选择器的设计，避免了三态的实现。与使用三态缓冲器的方法相比，这种互联的方法提供了更高的性能和更低的功耗。所有总线主设备维护自己的地址并且控制信号，以指示每

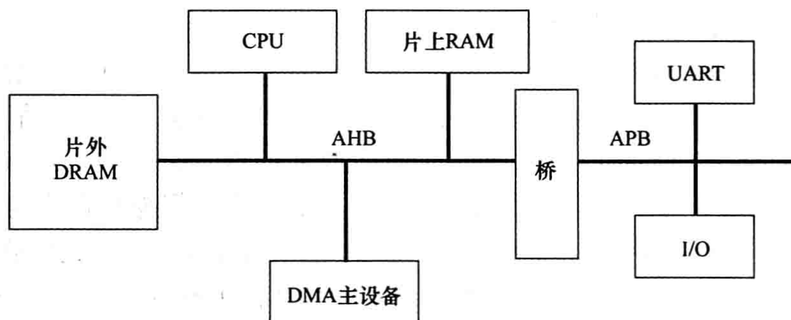


图 5.2 一个典型的 AMBA 总线系统[93]

个主设备所需传输的类型。一个中心仲裁器决定哪一个主设备拥有到达所有从设备的地址和控制信号。一个中心解码器电路选择正确的读数据然后返回来自参与事务的从设备的反馈信号。图 5.3 给出的三个主设备和四个从设备的多路选择互联就是这样一个多路选择器的互联设计。

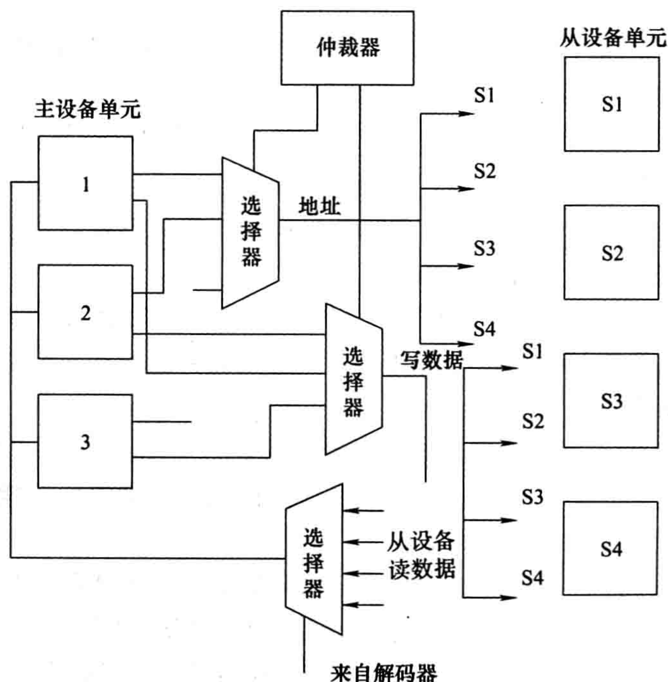


图 5.3 三个主设备/四个从设备的多路选择互联[22]

在 AHB 上的事务步骤如下：

- 总线主设备获得总线的访问权。这个过程以主设备发起一个请求信号到仲裁器开始。如果多于一个的主设备同时请求对总线的控制权，仲裁器决定哪一个主设备被授予对总线的使用权。

● 总线主设备初始化传输。被授予总线使用权的主设备，驱动地址和控制信号，控制信号包括地址、方向和传输的位宽。它也会检查是否本次传输是一个突发的一部分，以适用于突发传输的操作。一个写数据总线操作将数据从主设备移到从设备，而一个读数据总线操作将数据从从设备移到主设备。

● 总线从设备提供反馈。从设备返回给主设备的关于传输状态的信号，如传输是否成功、是否需要延时或者是否有错误发生。

图5.4a给出了一个基本的AHB传输。一个AHB传输由两个区别显著的过程段：地址过程和数据过程。主设备发起地址（ADDR）并在地址过程的时钟的上升沿控制信号，这常常持续一个周期。然后从设备采样地址和控制信号，并且在数据过程段，根据数据的读取（RDATA）和写入（WDATA）操作给出相应的反馈，以发送就绪（READY）信号作为完成的标志。从设备通过延迟发送（READY）信号，可以向任何传输加入等待状态，如图5.4b所示。对于一个写操作，总线主设备在整个数据周期中保持数据的有效。对于一个读传输，从设备不提供有效的数据直到数据过程段的最后一个周期。

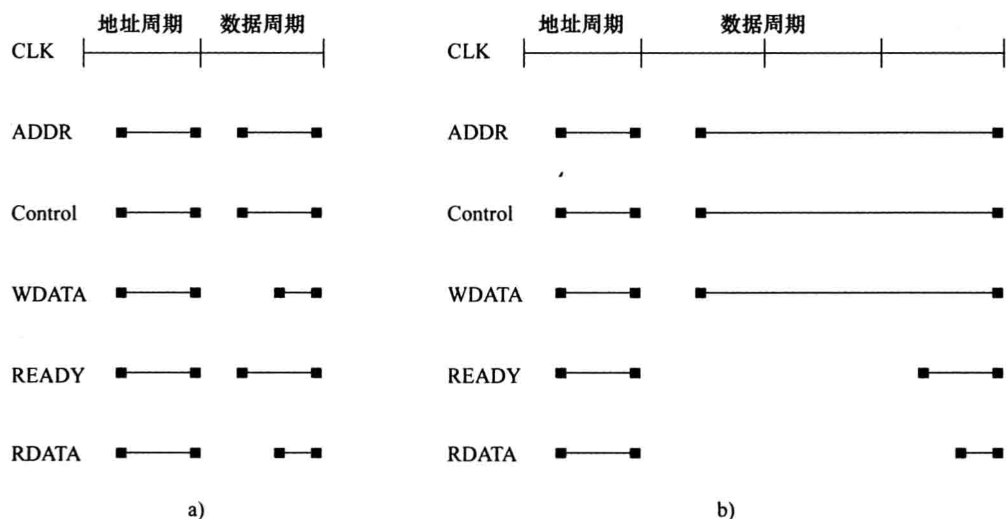


图5.4 一个基本的AHB传输[22]
a）传输中没有等待状态 b）传输中有等待状态

AHB 是一个管道式（独占）总线。因此，在前一个传输的数据过程段，任何传输的地址过程都可以发生。这种重叠的管道特点使得总线操作具有很高的性能。

APB 相比于性能，APB在最小功耗和低复杂度上更具优点。它被用于低带宽的外围设备的接口。

APB的操作是直接的，可以用三态的状态图来描述。APB或者保持空闲状态，或者在建立状态循环，或者处于数据传输使能状态。

5.4.2 CoreConnect 总线

和 AMBA 总线一样，IBM 的 CoreConnect 总线也是一个 SoC 总线标准。它被用于一个特别的处理器核——PowerPC 的外围设计，但它也用于其他的处理器。CoreConnect 总线和 AMBA 总线有相同的特点：它们都有一个层次化的总线，以满足不同级别的总线性能和复杂度；它们都有高级总线的特点，如多主设备、分离的读/写端口、管道技术、分离的事务、突发模式传输和扩展的总线位宽。

CoreConnect 总线的结构提供了三种总线用于核、宏单元库和自制逻辑的互联：

- 处理器本地总线（Processor Local Bus，PLB）
- 片上外围总线（On-chip Peripheral Bus，OPB）
- 设备控制寄存器（Device Control Regtster，DCR）总线

图 5.5 给出了一个典型的 SoC CoreConnect 总线系统，说明了在 SoC 中围绕 PowerPC 是如何利用 CoreConnect 总线结构的。高性能、高带宽块，如 PowerPC 440 CPU 核，PCI-X 总线桥和 PC133/DDR133（133 MHZ 总线的 DDR1）SDRAM 控制器，通过 PLB 被连接在一起，OPB 保持外围的片上低数据速率。菊花链 DCR 总线为在 PowerPC 440 CPU 核和其他片上模块这件传递配置信息和状态信息提供了一个相对低速的数据通路。

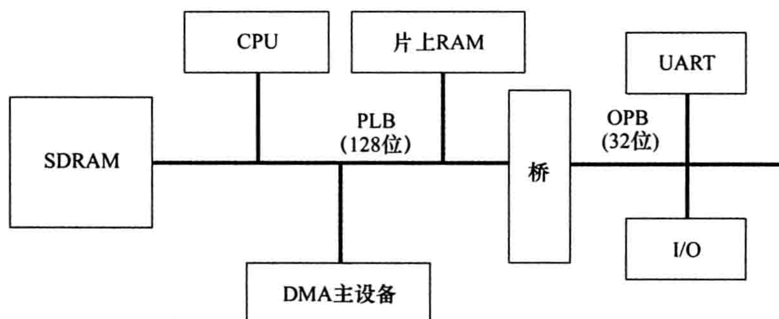


图 5.5 一个典型的 SoC CoreConnect 总线系统[118]

PLB PLB 用于高带宽、高性能和低延迟的处理器、内存和 DMA 控制器[118]之间的互联。它是完全同步的分离事务的总线，有分开的地址、读和写数据总线，允许两个传输每个时钟周期同时传输。所有主设备有它们自己的地址、读数据、写数据和控制信号，这些信号叫做传输修饰信号。总线从设备也具有地址、读数据和写数据总线，但是这些总线都是被共享的。

和 AMBA AHB 一样，PLB 事务有多个阶段组成，这些阶段可能会持续一个或者多个时钟周期，并且地址和数据总线是分离的。事务的地址总线有三个阶

段：请求（RQ）、传输（XFER）和地址反馈（ACK）。一个 PLB 的事务开始于一个主设备驱动它的地址和传输控制信号，并且请求在地址独占请求阶段时对总线的占有权。一旦 PLB 仲裁器将总线占有权授予主设备，在传输阶段，主设备的地址和传输信息会被呈现给从设备。在地址反馈收到阶段，地址周期在从设备锁存主设备地址和传输信息后结束。

图 5.6 给出了一个描述了两个深度读和写地址的管道传输和并发读和写数据时的独占总线情况。主设备 A 和主设备 B 代表每一个主设备的地址和传输信息的状态。PLB 在这些请求中进行仲裁，并将选择的主设备的请求传递给 PLB 从设备的地址总线。标识着地址阶段的通路展示了 PLB 从设备地址总线在每一个 PLB 时钟下的状态。

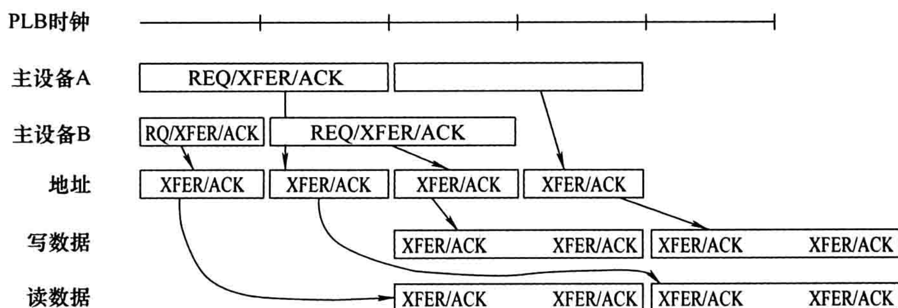


图 5.6　一个 PLB 传输协议[118]

在数据独占总线阶段的每一次数据传递有两个阶段：传输和反馈收到。在传输阶段，主设备驱动写数据总线进行写传输或者采样读数据总线进行读数据传输。如图 5.6 所示，第一个（或者只有一个）数据写传递和地址传输阶段同时发生。

分离事务　PLB 地址、读数据和写数据总线被减弱，以允许地址周期被重叠使用于读和写数据周期，并且允许读数据周期被重叠用于写数据周期。PLB 分离总线事务的能力允许地址和数据总线在同一时刻有不同的主设备。另外，第二个主设备可能在另一个主设备总线的数据传输周期内，通过地址管道请求对 PLB 总线的占有权。这种情况利用图 5.6 所示的通过各种信号之间的依赖来描述，信号用箭头表示。

OPB　OPB 是第二种总线设计，它通过减少 PLB 的承载容量[126]来减轻系统性能的瓶颈。适用于 OPB 的外围设备包括串行端口、并行端口、UART、通用目的 I/O（General Purpose I/O，GPIO）、计时器和其他低带宽的设备。OPB 比 AMBA APB 更加复杂。通过将地址和数据总线设计为分配选择器，它支持多主设备和从设备。这种结构适用于数据不敏感的 OPB，并且允许外围设备加入到自制的核心逻辑设计中而不改变 OPB 仲裁器或者已存在的外围设备的 I/O。图 5.7 给出了一种 OPB 结构，其中包括了地址和数据总线结构。主设备和从设备都为外出总线提供使能控

制信号。通过要求每一个单元提供这个信号，相关联总线的组合逻辑可以被有策略地分布在整个芯片上。如图 5.7 所示，任何一个主设备有向从设备提供地址的能力，然而主设备和从设备都具有驱动和接受分布式数据总线的能力。

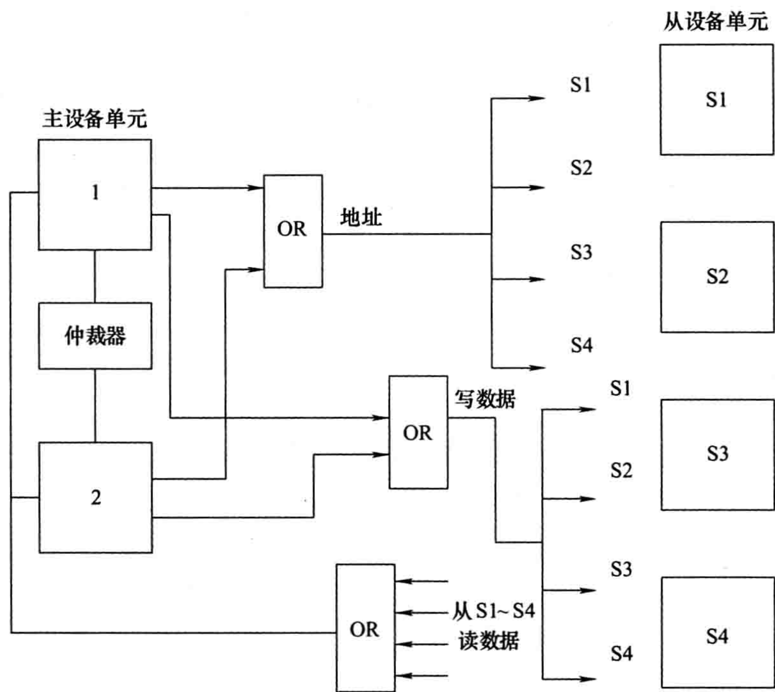


图 5.7　一种 OPB 结构[126]

表 5.3　给出了 CoreConnect 和 AMBA 总线标准的比较。

表 5.3　CoreConnect 和 AMBA 总线标准的比较[198]

	IBM CoreConnet PLB	ARM AMBA 2.0 AMBA 高性能总线
总线结构	32 位、64 位和 128 位，可扩展到 256 位	32 位、64 位和 128 位
数据总线	分离的读和写	分离的读和写
主要性能	多总线主设备	多总线主设备
	四深（four-deep）读管道，二深（two-deep）写管道	管道技术
	分离事务	分离事务
	突发传输	突发传输
	线传输	线传输
	OPB	AMBA APB
主设备支持	支持多主设备	单一设备：APB 桥
桥功能	PLB 或 OPB 上的主设备	只有 APB 主设备
数据总线	分离的读和写	分离的读和写

5.4.3 总线接口单元：总线套接字和总线封装

让不同的可重用的IP核综合使用一个标准的SoC总线，有一个重大的缺点。因为标准总线在线路连接上定义协议，一个通过一种总线标准编译的IP核不能使用在用其他总线标准编译的块中。解决这个问题的一个方法是采用一个硬件"套接字"，这是5.2节中总线封装的一个例子，它通过使用良好设计的不依赖物理总线协议的IP核协议，将互联逻辑和IP核进行分离。因此，核对核的通信被接口封装处理。这种方法被虚拟套接字接口联盟（Virtual Socket Interface Alliance，VSIA）[42]通过它们的虚拟组件接口（Virtual Component Interface，VCI）而使用，并且也被美国Sonics公司通过采用开源协议（Open Core Protocol，OCP）和Silicon Backplance μNetwork所使用。

VSIA提倡一系列的标准和接口，被熟知的是虚拟套接字接口（Virtual Socket Interface，VSI），它使得芯片上系统级的互动能够使用预先设计好的模块（称作虚拟组件㊀）[247]。这鼓励了使用组件范式进行设计。VC是高效 的IP模块，符合VSI的规定，它们可以是三种类型中的一种。硬VC由所有定义的硅层上布局布线好的门组成。它的性能、面积和功耗是可预测的，但不够灵活。软VC被定义在硬件描述语言的表示中，它们通过综合、布局和布线被映射到物理设计中。它们可以被方便地修改，但是通常需要更多的精力在SoC设计中进行综合和验证，同时它的性能可预测性也较小。最后，固件VC在这两者之间进行了折中。它们以发生器或部分布局好的模块库的形式，通过要求最后的布线或者布局调整来实现。这种形式的VC比软VC提供了更加可预测的性能，在一些情况和配置下仍旧是比较灵活的。

为了将这些不同的VC连接起来，VSIA研发出一个VCI的规定，在这个规定下其他合适的总线可以与其进行接口连接。根据VCI的规定，设计者可以将一个VC和其他任何几个总线综合使用，从而达到系统性能的要求。VCI标准定义了一系列的协议。目前定义了三种协议：外围VCI（Peripheral VCI，PVCI），基本VCI（Basic VCI，BVCI）和高级VCI（Advanced VCI，AVCI）[247]。PVCI是一个低性能协议，它的请求数据和反馈数据的传输发生在一个单一控制握手事务中。因此，它不是分离事务的协议。BVCI采用分离事务的协议，但是反馈必须按顺序到达。换句话说，反馈数据必须和产生请求的顺序一致。AVCI和BVCI相似，但是允许乱序的事务。请求被加上标签，并且事务可以交叉进行和重新排序。

另外，VSIA也定义了一些抽象层表示，用于将VC集成到SoC设计中[42]。

㊀ 虚拟组件：Virtual Component，VC。

这样做的思想是，如果 IP 模块提供者（VC 提供者）和系统综合者（VC 综合者）符合 VSI 在所有抽象层的规定，那么使用 IP 组件范式的 SoC 设计就可以在较低错误风险下进行。

相对于 VCI 的另一个选择是 OCP，它被国际开源协议组织（Open Core Protocol International Partnership，OCP-IP）所推荐[188]。OCP 在两个通信的实体之间定义了一个点对点的接口，如在两个 IP 核之间使用核中央协议。使用 OCP 的接口假定具有套接字的属性，如之前所说的，它是一个高效的总线封装，允许对目标总线进行接口连接。图 5.8 给出了一个使用 OCP 和总线封装的三 IP 核模块系统。一个模块是系统初始化器，一个是系统目标，另一个既是初始化器也是目标。

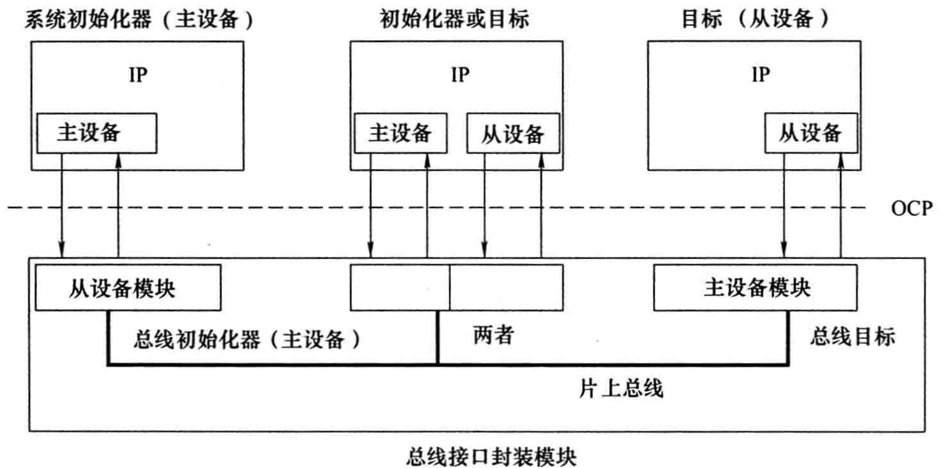


图 5.8 一个使用 OCP 和总线封装的三 IP 核模块系统[225]

另一个互联层次可以在 OCP 的基础上进行，从而有助于进行更深入的 IP 综合。Sonics 公司推崇它们专有的 SiliconBackplance 协议，这个协议无缝地将使用 OCP 的 IP 模块连接起来。不同模块之间的通信在 Silicon Backplane μNetwork 上进行，这个 μNetwork 具有可扩展的 50 ~ 4000 MB/s 的带宽。图 5.9 给出了美国 Sonics 公司的 μNetwork 配置，说明了组件是如何连接在一起的[225]。

使用封装技术的总线接口单元已经说明了如何减少设计 SoC 的时间，但是它是建立在开销门资源和延迟的基础上的。加入简单的封装硬件会增加存取的延时，并且会导致 3 ~ 5K 个门的硬件开销[160]。

另外，总线接口单元可以使用 FIFO 缓冲器来提高性能。图 5.10 给出了在总线接口单元采用写数据缓冲器所带来的硬件开销和性能的提升。

写缓冲器提供了几个周期的延迟优化和 10% 的吞吐量增加，延迟取决于数据的大小。

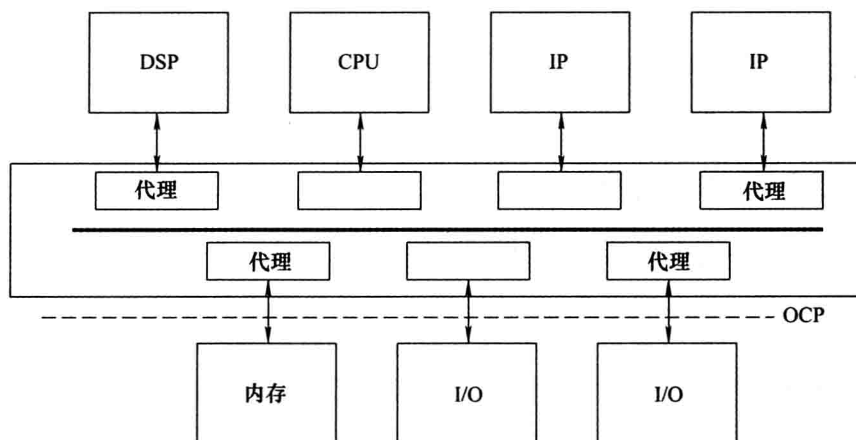


图 5.9 美国 Sonics 公司的 μNetwork 配置[225]

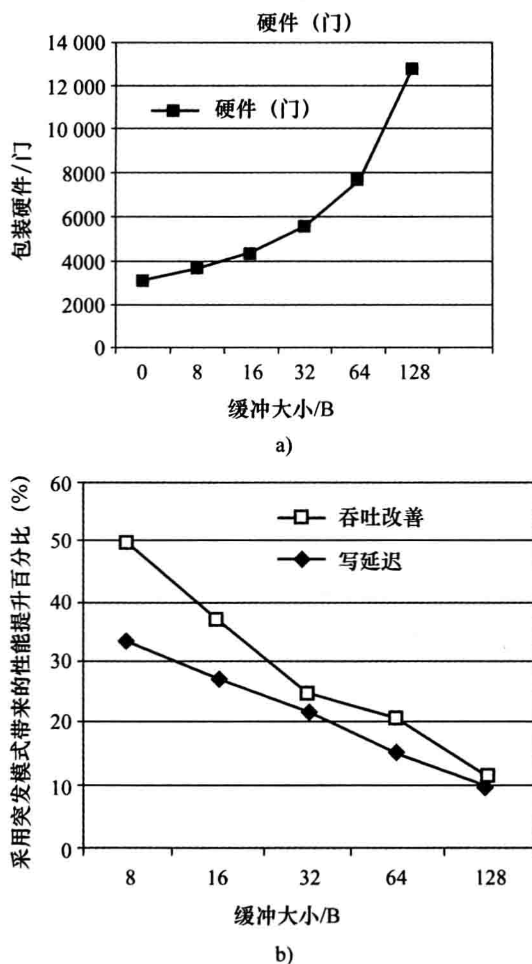


图 5.10 在总线接口单元采用写数据缓冲器所带来的硬件开销和性能的提升[8]

a）写缓冲器的硬件开销 b）缓冲器对突发模式传输的性能影效

5.5 总线模型分析

5.5.1 竞争和共享总线

两个或多个单元同时请求一个不能被提供的共享的资源时，就发生了竞争。竞争发生时，有两种处理方式：（1）延迟请求并置为空闲状态直到资源可用；（2）在一个缓冲中对请求进行排队直到资源可用时再开始运行。方式（2）只在请求的资源对于程序运行不是逻辑上必须的情况下发生（如缓存的预取）。

是否需要分析总线的竞争取决于总线的最大（或提供的）带宽和内存带宽的关系。竞争和排队是系统的瓶颈，因此，最被限制的资源是具有竞争性的资源和系统的延迟部分资源。因此，当总线比内存受到更大限制时（具有较小的可用带宽），总线必须进行竞争分析。

总线常没有缓冲（队列），存取延时导致系统变慢。总线阻塞的影响分析取决于存取类型和缓冲技术。

通常有以下两种类型的存取模式：

1. 没有立即重新提交的请求。被拒绝的请求返回和原始请求一样的到达分配。一旦请求被拒绝，程序继续执行，尽管重新提交请求会产生延迟。缓存行的预取就是这样一个例子，它不被当前程序的继续执行所需要。

2. 立即重新提交的请求。当多个独立的处理器存取一个共同的总线时，这是更加典型的例子。程序不能在一个请求被拒绝后继续执行。被拒绝的请求会立即重新提交。处理器处于闲置状态直到请求被接受并执行。

5.5.2 简单的总线模型：没有重新提交

下面，假定每一个请求占据总线相同的服务时间（即 $T_{\text{line access}}$）。即使有两个不同类型的总线使用者（即，单线上的字请求和线请求或［脏］双线上的请求），大多数情况可以通过简单的计算每一个处理器的平均总线占有率 ρ 来合理地估计，公式如下：

$$\rho = \frac{\text{总线传输时间}}{\text{处理器处理时间} + \text{总线传输时间}}$$

处理器时间是处理器在处理一个总线请求之前的所需计算的平均时间。当然，处理器可能将一些计算时间与总线时间重叠。在这种情况下，处理器时间是总线请求之间不重叠的时间。任何情况下，$\rho \leq 1$。

n 个处理器访问总线的最简单模型如下：

$$\text{Prob}(\text{处理器不访问总线的概率}) = 1 - \rho$$

$$\text{Prob(总线忙的概率)} = (1-\rho)^n$$
$$= \text{宽带的分数形式} = B(\rho,n)$$

带宽的分数形式乘以最大总线带宽，得到实现的总线带宽 Bw。

每个处理器的实现带宽的分数形式（实现的占有率）ρ_a 由如下公式给出：

$$n\rho_a = B(\rho,\ n)$$

$$\rho_a = \frac{B(\rho,\ n)}{n}$$

由于总线阻塞，处理器变慢，为 ρ_a/ρ。

5.5.3 重新提交的总线模型

支持请求重新提交的模型需要更加复杂的分析和反复的解决办法。这里有几个解决方法，每个方法都产生相似的结构。被 Hwang 和 Briggs[118] 提出的解决方法是对下面一对方程的反复迭代：

$$a = \frac{\rho}{\rho + (\rho_a/\rho)(1-\rho)}$$

和

$$n\rho_a = 1 - (1-a)^n$$

式中，a 为实际提供的请求率。为了得到 ρ_a，初始化时以 $a=\rho$ 作为迭代的开始。一般四次迭代后会收敛。

5.5.4 使用总线模型：计算给定的占有率

在执行部分中的模型在事务类型中没有区分。它仅要求总线事务的平均时间，这个时间是总线处理事务所用的平均周期数。下一个问题是找到给出的占有率 ρ。

给出的占有率是在事务之间没有竞争的情况下总线繁忙时间的分数部分。为了找到这个分数部分，需要决定总线事务的平均时间和事务之间的计算时间。

处理器初始化事务的自然性是另一个影响因素。简单的处理器产生阻塞事务。在这种情况下，处理器在总线请求产生之后处于空闲状态，并且仅在总线事务完成后重新开始计算。对于更加复杂的处理器的一个选择是缓冲的（或者没有阻塞的）事务。在这种情况下，处理器在作出一个请求后继续进行工作，并且确实可能在完成初始的请求之前产生几个请求。根据系统的配置，有两种常见的情况：

1. 具有阻塞事务的单一总线主设备。在这个例子中，没有总线竞争，因此处理器会等待事务的完成。这里，实现的占有率 ρ_a 与给出的占有率相同，并且 $\rho=\rho_a=$（总线事务时间）/（计算时间 + 总线事务时间）。

2. 具有阻塞事务的多总线主设备。在这个例子中，给出的占有率是 $n\rho$，ρ

是情况 1 中的ρ。现在会发生竞争，因此使用总线模型去决定实现的占有率ρ_a。

例 5.2

假设一个处理器有由缓存行传输组成的总线事务。假定 80% 的事务移动单行并且占据总线 20 个周期，20% 的事务移动双行（如脏行替换）并占据总线 36 个周期。总线事务的平均时间是 23.2 个周期。现在假设，每 200 个周期发生一次缓冲未命中（事务）。

在情况 1 中，总线被占用：$\rho = \rho_a = 23.2/223.2 = 0.10$。这里没有竞争，但是总线造成系统速度变慢，这在下面会讨论到。

在情况 2 中，假设有四个处理器。现在给出的占有率是$\rho = 0.104$并且用模型去计算竞争时间。开始时设置$a = \rho = 0.104$，$n\rho_a = 1-(1-a)^n = 1-(1-0.104)^4$；现在得到了$\rho_a$，然后用$\rho_a$替代$a$并继续。

因此，开始时，$\rho_a = 0.089$；下一次迭代后，$\rho_a = 0.010$；然后经过几次迭代后，$\rho_a = 0.095$。总是得到比给出的小的值，产生这种不同是由竞争带来的延迟造成的。因此有

$$\rho_a = 0.095 = \frac{\text{总线传输时间}}{\text{计算时间} + \text{总线传输时间} + \text{竞争时间}}$$

计算竞争时间，得到的结果是 21 个周期。

5.5.5 总线事务的影响和竞争时间

总线延迟对于整个系统的性能的影响有两个方面：第一个显然是阻塞，它简单地将事务延迟插入到程序的执行过程中；第二个影响是竞争，它降低了事务在总线和内存上流通的速率。这多少降低了系统的性能。

在阻塞的情况下，处理器因为总线事务数量较多而变慢。因此，与没有总线事务的理想处理器相比，其相对性能为

$$\text{相对性能} = \frac{\text{计算时间}}{\text{计算时间} + \text{总线传输时间}}$$

在情况 1 中，处理器变慢，为之前的 200/223.3 = 0.896。

当出现竞争是，会增加额外的延迟。在情况 2 中，单个处理器变慢为之前的 200/(223.2 + 21) = 0.819。竞争导致的系统（没有竞争）变慢以比率ρ_a/ρ来表示。由于这个比率，将减少提供的事务。

5.6 超越总线：拥有交换互联的 NoC

虽然总线互联在 SoC 互联结构中占主导地位，它仍具有很多缺点。即使一个设计很好的基于总线的系统，也可能遇到数据传输的瓶颈，也就限制了整个系统的性能。它也不具备可升级性。当越多的模块被添加到总线时，不仅数据阻塞增

加，而且功耗也会由于总线电路增加的负载而增加。基于交换的 NoC 互联避免了一些这样的限制。然而，交换器天生比总线要复杂，这在大型 SoC 配置中更有用。交换设计中有很大权衡的可能。大量节点可以用相对较小的延迟互联起来；但是在开销以指数增加的基础上（像交叉开关），也可用相对较大的延时互联起来，这时开销较小（如分布式互联）。

本节介绍了一些基本的概念和物理互联网络设计的一些方法选择。这个网络由配置的交换器组成，它将 N 个单元连接起来。互联网络设计的效率或者费用-性能高低由以下内容决定：

1. 将一个请求单元与它的目的单元连接起来的延迟。
2. 单元之间的带宽和连接的并发数量。
3. 互联网络的开销。

在一个网络中，单元之间通过链路或者通道来通信，通信方式可以是单向的也可以是双向的。链路具有带宽或每个单元时间的传输字节数，从而能够在单元（节点）之间并行地进行传输。一个节点的分列数是连接到邻近节点的双向通道的数量（见图5.11）。

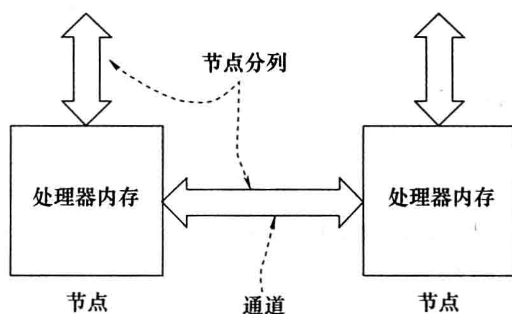


图5.11 节点和通道（节点的分列数是连接到邻近节点的通道数量）

网络可以是静态的或是动态的。在一个静态网络中，拓扑结构或者节点之间的关系是固定的（见图5.12）。两个节点之间的通路不会改变。在一个动态网络中，可以选择节点之间的通路，以实现连接性和提高网络的带宽（见图5.13）。

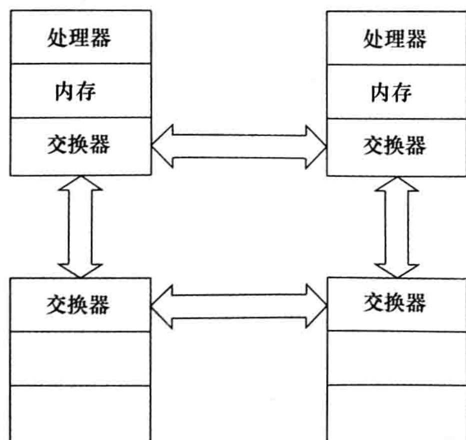


图5.12 静态网络（单元之间的连接是固定的）

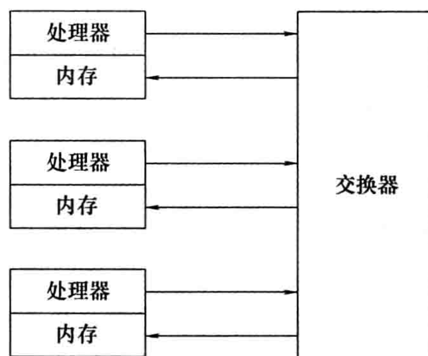


图5.13 动态网络（单元之间的连接多变，形成连接）

静态网络可以包含一个二维网格的交换器，从而将 SoC 的模块连接起来。动态网络可以由一个中心交叉开关构成。除了可以避免传输阻塞的优点之外，基于交换的设计可以允许模块在不同的时钟频率下工作，并且可以减轻总线负载问题。

图 5.14 给出了基于交换的互联，它将同一芯片本地的一些同步模块连接在一起[60]。这个交叉开关是完全异步的。在芯片中，时钟域交换器被用于向同步模块桥接异步互联。

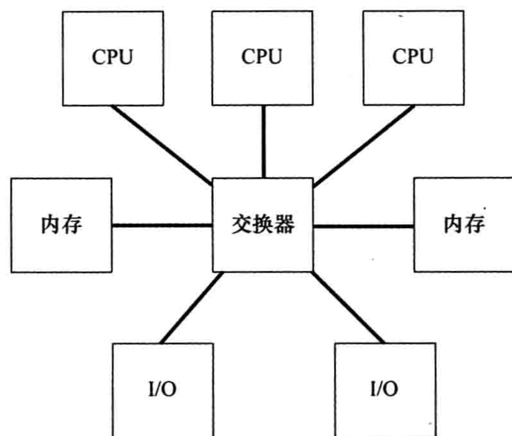


图 5.14　基于交换的互联

SoC 互联交换

本节对来自计算机互联文献中一些基本概念和结果进行总结。在 SoC 交换中，目前节点（单元）由于模具大小被限制在 16 ~ 64 个。由于单元在片上，连接带宽 w 相对比较大：16 ~ 128 条线。在 SoC 中，目前动态网络占据主导地位（纵横开关或者多级连接）；使用的静态网络为网格状（环面）。随着 SoC 单元数的增加，期望可以实现更加多样的互联网络。

5.6.1　静态网络

在静态网络中，两个单元之间的距离是为了建立它们之间的通信必须要通过的链路或通道（或跳）的最小数量。网络的直径是两个任意网络单元之间的最大距离（没有重复通路）。一个静态网络的线形网络如图 5.15a 所示。网络可以是开放的，也可以是自闭的。一个自闭的网络通过将一个线形阵列交换为环，提高了平均距离和直径（见图 5.15b）。最常见的静态网络类型是（k，d）网络[66]。这是由维数为 d，节点数为 k 的阵列重构形成的。这种网络常是自闭的，如环，它的 $d=1$；如网，它的 $d=2$。

假定在一个线形阵列中有 k 个节点，希望去扩展网络。可以增加网络的维数，产生一个二维的网格网络，其中 $d=2$（见图 5.15c），而不是简单地增加线形元素数量。这种（k，d）网络可以是，线形阵列，$d=1$；二维网格，$d=2$；立方阵列，$d=3$；或超立方体。超立方体结构常将每维的元素数量限制在两个，即 $k=2$，使用尽可能多的维来包含整个网络。高维网络提高了连接性，但是增加了连接交换的费用。在每一个最邻近的单元之间必须有一个交换器，在（k，d）网络中通常有 $2d$ 个邻紧单元。图 5.15d 给出了一个环，常被称作最邻近单元的网状阵列。

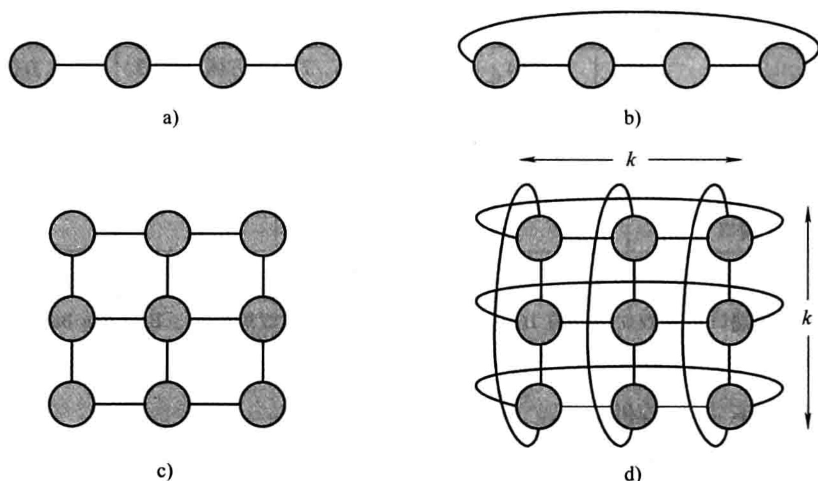


图5.15 没有预选位置的静态网络例子

（这些也被称作(k, d)网络。在图a和b中，令$k=4$，$d=1$（一维）；在图c和d中，令$k=3$，$d=2$） a）线形阵列 b）含有闭包的线形阵列（环） c）网格（二维网） d）包含闭包的$k\times k$网格（二维网）

特别的，在二元立方体结构或者超立方体结构中，$k=2$。超立方体的节点数N和维数按如下方法决定：对于（2，n）网络，有双向通道的二元n方体有

$$N=2^n$$

对于（2，n）情况，有

$$\text{直径}=n$$

对于一般的具有n维，闭包和双向通道的（k，n）网络，有

$$N=k^n$$

或者

$$n=\log_k N$$

和

$$\text{直径}=\left[\frac{k-1}{2}\right]n$$

例5.3 假设有一个4×4的网格（图5.15d所示的环）。按（k，d）来说它是一个（4，2）网络，$N=16$，$n=2$，并且维数为4。

通常，网络的维数和它最大的距离对于开销和性能是重要的。表5.4显示了一些关于64节点（$N=64$）静态（k，d）网络开销和性能的比较。

表5.4 一些关于64节点（$N=64$）静态（k，d）网络开销和性能的比较

	环（64，1）	网格（16，2）	立方体（4，3）	超立方体（2，4）
性能				
跳数（平均，$dk/4$）	16	8	3	2

（续）

	环（64，1）	网格（16，2）	立方体（4，3）	超立方体（2，4）
直径（跳数）（最大内部节点距离，$dk/2$）	32	16	6	4
开销				
节点分列（端口），$2d$	2	4	6	8
双向 BW，$2\ wN/k$	32	128	512	1024

注：链路（和端口）是双向的，有 16 条线（$w=16$）。BW（双段带宽，Bisection bandwidth）指当网络被分为两个相同的部分时，被分为两段的信号线的量。

链路有以下三个特性：

1. 链路的周期时间 T_{ch}。它指信息在邻近节点传播的时间。$1/T_{ch}$是链路或通道的线路带宽。

2. 链路的位宽 w。它决定在两个节点之间能够被同时传输的位的数量。

3. 不区分链路是单向的还是双向的。

与链路特性符号相联的是信息的位长度 l 加上 H 位头部。这个头部仅是目标节点的地址。因此，$T_{ch} \times (l+H)/w$ 是在两个邻接单元之间传输一条信息的时间。

假设节点 A 有一个发送到节点 C 的信息，该信息必须通过节点 B 进行传递。如果节点 B 是可用的，这条信息首先从 A 传输到 B，然后在 B 保存。在信息被完全传输后，节点 B 访问节点 C，如果节点 C 可用，则将该信息传递给节点 C。与其将信息存储在节点 B，还可以用虫孔路由[70]的方法。当信息在节点 B 被接收后，信息仅被暂存足够用于解码头部和决定目的地长度。一旦目的地被决定，这条信息会被重新传输到节点 C，前提是节点 C 是可用的。在节点 B 所需要的缓冲空间会大幅度缩减，并且传输的总时间为

$$T_{wormhole} = T_{ch}(d \times h + l/w)$$

式中，$h = [H/w]$。

例 5.4 在一个 4×4 网格，$(k, d) = (4, 2)$，并且假定 $T_{ch}=1$，$h=1$，$l=256$，$w=64$。则 $T_{wormhole}$ = (2+4) 个周期 =6 个周期。

一旦头部在一个中间的节点被解码，这个节点可以决定这个信息是要向那个节点发送的。这个中间节点选择一个到目的节点最小距离的通路。如果多个通路具有相同的长度，则中间节点会选择当前不阻塞或可用的那个通路。

5.6.2 动态网络

一个基本的动态间接交换网络如图 5.16 所示。

一般而言，动态网络中的基本元素是交叉开关（见图 5.17）。

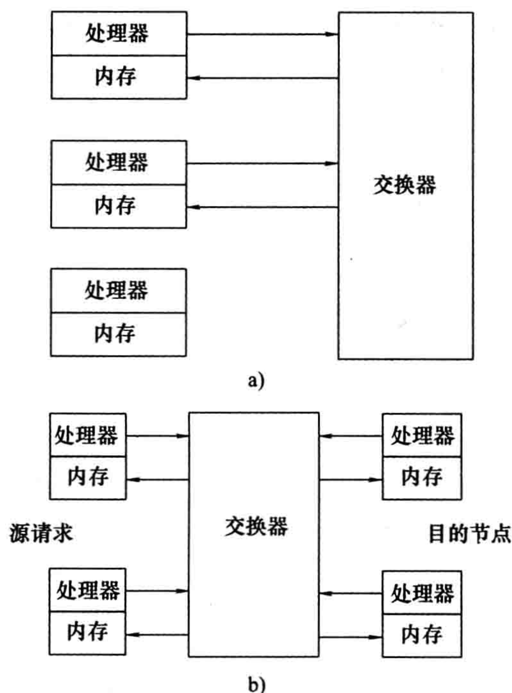


图5.16 一个基本的动态间接交换网络

a）一个集中式的交换网络（它与处理器分离） b）一个分布程度更大的网络

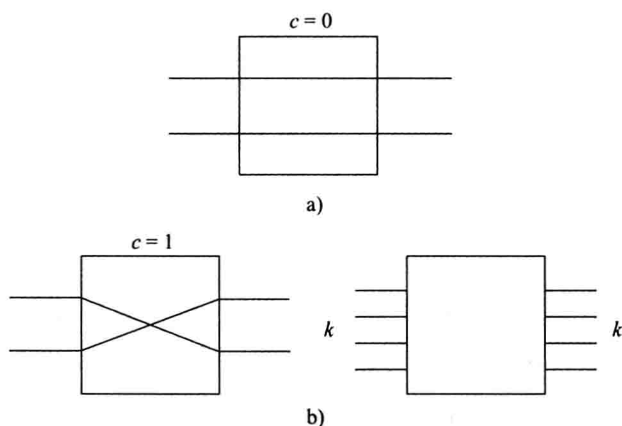


图5.17 交叉开关

a）一个 2×2 的具有控制信号 c 的交叉开关 b）它可以生成 $k \times k$ 的交叉开关

交叉开关将 k 个点与另外 k 个点相连接。只要任意两条信息没有相同的目的地，多个信息就可以并行地穿过交叉开关。交叉开关的费用以 n^2 的形式增长，因此对于大网络，使用交叉开关费用过高。为了控制交换器的开销，可以使用一个小的交叉开关作为多级网络的基础，这种方法常被称为多级互联网络（Multistage Interconnection Network，MIN）[256]。

有很多种类的MIN，包括基线、Benes、Clos、Omega[150]和Banyan网络。基线网络是最简单的，如图5.18所示。

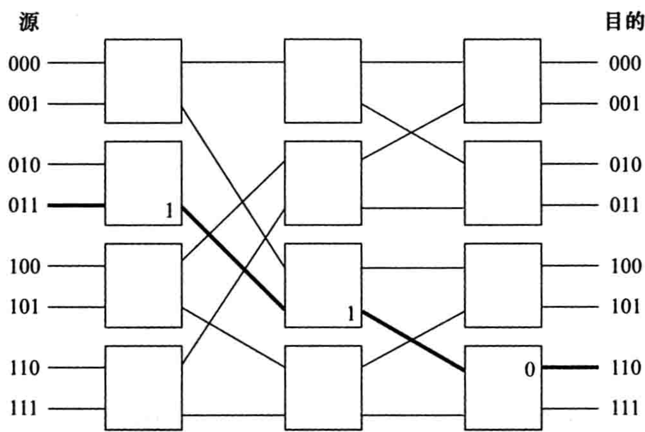


图5.18　基线动态网络拓扑结构

头部信息可以引起连续级的交换，从而正确的连接通路可以在两个节点之间建立。例如，考虑一个 M，N 网络的确定的布线算法。假定节点011向目的节点110发送一条信息。交换器从1、1和0输出，从而使信息达到110目的节点，通过设置控制信号 c，无论上部的"0"或者底部的"1"，都可以被交换器选择。相似的，返回通路是011。两个节点之间的阶段数为

$$阶数 = [\log_k N]$$

式中，k 为到达交叉开关元素（$k \times k$）的输入数量。因此，一位宽度的通路所需的（$k \times k$）交换器的总数为

$$\frac{N}{k} \times [\log_k N]$$

其他动态网络在信息带宽的实现，信息延迟和容错方面做出了不同的权衡。表5.5给出了一些常见动态网络的特点。

表5.5　一些常见动态网络的特点（使用 $k \times k$ 交换器交换N输入×N输出）

网　　络	其他等价的网络	延迟阶段（单元中的 $k \times k$ 交换器延迟）	阻　　塞	开销估计（$k \times k$ 交换器）
Baseline	Delta，Omega，SW Banyan	$[\log_k N]$	有阻塞	$\frac{N}{k}[\log_k N]$
Benes	—	$2[\log_k N]-1$	如果配置好，不会阻塞	$\frac{2N}{k}[\log_k N]$
Clos	—	$2[\log_k N]-1$	完全不阻塞	$\frac{4N}{k}[\log_k N]$

5.7 一些 NoC 交换的例子

5.7.1 直接网络的一个二维网格的实例

通过将用户的 IP 核用直接互联网络连接起来，数据传输被分布在整个 NoC 中。数据传输的瓶颈被避免，因为在节点之间有很多条通路，并且数据传输可以同时进行。Xfabric 方式利用一个二维网格直接网络，将用户的核心在 Xilinx FPGA 上连接起来，如图 5.19[60] 所示。拥有一个到四个通信端口的数据处理核通过一个交换组件网络被连接起来（图中灰色部分）。这些数据路由连接自主地管理了系统多个用户核之间的数据流。多个连接的实例组成了直接二维网格网络，它可以互联 1024 个单端口核心。在交换组件之间的水平和垂直数据传输链路使得核之间的数据通信更加高效。

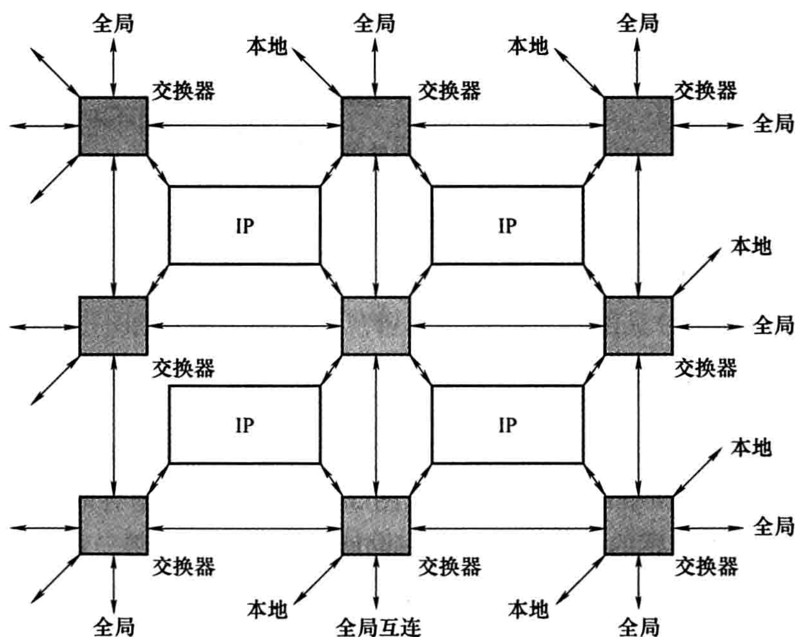


图 5.19 通过交换组件以 Xfabric 方式连接数据处理器[64]
（它是使用二维网格拓扑结构的直接交换网络）

图 5.20 给出了交换组件原理图。每个交换器有四个本地端口（LPORT0 ~ LPORT3）和四个全局端口（GPORT0 ~ GPORT3）组成。用户核通过本地端口来发送 48 位字和接收 32 位字，16 位的全局端口用于将数据路由到邻近的交换器。

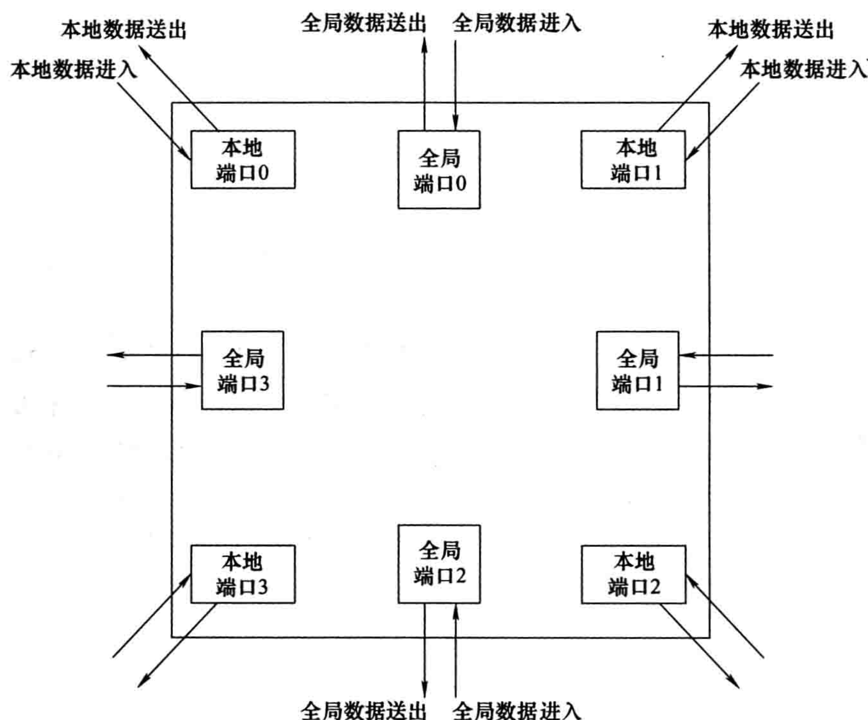


图 5.20 交换组件的原理图[60]

每个交换组件都具有路由和仲裁的功能，它可以将多个并行数据流在数据源单元和目的单元之间以最小的延迟传送，因此避免了基于总线的系统中的传输瓶颈。

5.7.2 同步 SoC 的异步交叉互联（动态网络）

用于 SoC 应用的另一个 NoC 是美国 Fulcrum 公司设计的 PivotPoint 结构[66]。这个系统的中心是 Nexus 交叉开关（见图 5.14），它的数据吞吐率为 1.6Tb/s。Nexus 使用时钟不敏感的异步电路。它在设计风格上具有优势，包括程序的自适应技术、环境的多变性和较低的系统功耗。对多时钟域核互联的需求是选择异步设计风格的原因。异步核心可以在不同的时钟频率下工作，并且具有各自独立的相位。在同步核心和异步交换矩阵之间需要时钟域交换器进行接口连接。由于交叉开关不使用时钟信号，综合不同的时钟域不需要额外的开销。这样，系统是全局异步的，但在本地是同步的，这被称作 GALS 系统。

在 Nexus 上的数据传输通过突发来完成。每次突发包含数量可变的数据字，突发通过一个尾部信号来结束。一个 4 位的控制信号被用于表示目的通道（TO），它在突发离开交换器时变为源通道。Nexus 上的突发格式如图 5.21 所示。突发通过交换器自动路由，并且它不能中断、分段或者复用。

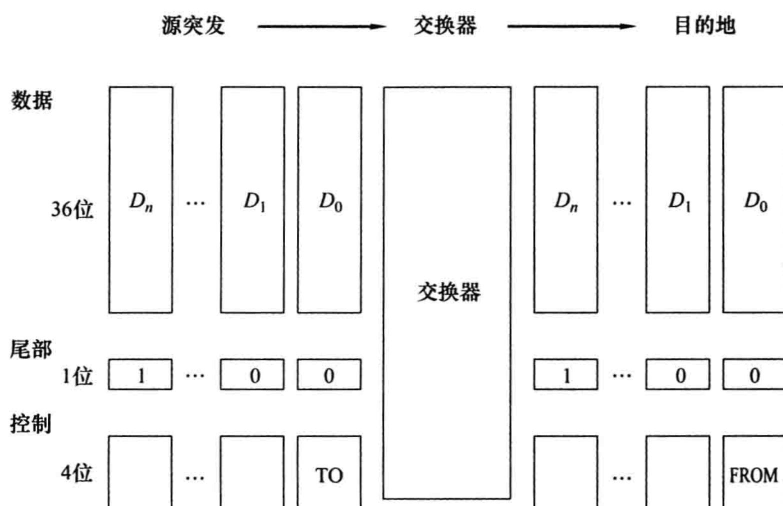


图 5.21 Nexus 上的突发格式

交换器在物理链路上为突发提供通路，通路在突发的第一个字进入交换器时被创建，在最后一个字离开交换器时被关闭。

5.7.3 阻塞与不阻塞比较

Nexus 和 PivotPoint 被设计用于避免队头（Head Of Line，HOL）阻塞。当一个信息包不能被传递而导致后面的不相关信息包阻塞时，就会导致 HOL 阻塞发生。PivotPoint 使用虚拟通道（也称作端口）同时传输分开的信息流。被阻塞的信息包仅阻挡同一通道中之后的信息包。其他通道的信息包可以自由前进。这样，通信的停顿被减到最小。

5.8 分层结构和网络接口单元

网络接口单元是 NoC 中的关键组件，因为它可以解决一系列传统的基于总线的方法[34]的限制。尽管之前讨论的总线标准提供了一定程度上的 IP 核的便携性和可重用性，它们很难调整为高级的程序和总线接口技术。总线的基本缺点是它们不使用分层的方法来进行互联。在应用层的事务通信级和物理层的信号互联之间没有明显的分离。相反，在 NoC 系统中的活动通常被分离成事务层、传输层和物理层，如图 5.22 所示。因此，NoC 系统可以很好地适应高级程序技术和高级系统结构的快速发展。

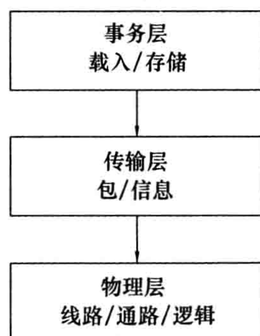


图 5.22 NoC 的三层结构[25]

图5.23显示了一个通常的片上互联网络，它由一些模块组成，如处理器，内存和IP模块。这些模块被连接到网络上，网络可以建立模块之间数据传输的通路。模块之间的所有通信通过网络进行，并且网络逻辑的面积使用减少为6.6%[71]。这种NoC结构的重要特点是：（1）分层结构可以很容易的升级；（2）灵活的交换拓扑结构可以被用户设置，从而优化性能和适用于不同的应用；（3）点对点通信有效地减弱了IP模块之间的信息阻塞。

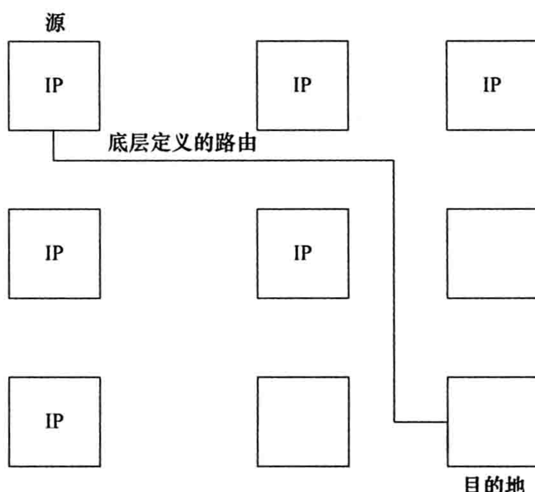


图5.23 一个典型的NoC结构[25]

5.8.1 NoC的分层结构

大多数的NoC结构采用一个三层的通信结构，如图5.22所示。物理层定义了信息包如何在物理接口传输。程序技术、互联交换结构和时钟频率的任何改变都会影响到这个层。更往上的层则不会受到影响。

传输层定义了信息包如何在交换网络上路由。信息包中头部的一个小部分被用来指明路由如何完成。事务层定义了用于将IP模块连接到网络的原始通信。NoC接口单元（NoC Interface Unit，NIU）向IP模块提供了事务层的服务，以保证信息在NoC接口之间的交换，从而实现了一个特定的事务（见图5.24）。

NoC的层次结构有如下好处：

1. 物理层和传输层可以被分开优化。物理层被程序技术所支配，事务层则依赖特定的应用。分层的方法允许它们在互不影响的情况下分开优化。

2. 固有的可升级性。NoC中一个正确设计的交换结构可以被升级，以解决任何数量的并行事务。该结构的天然分布性允许交换器进行优化，以适应需求。同时，负责事务层的NIU经过设计，可以满足IP模块性能的需求，使得IP核不用在配置和交换结构的性能上花费精力。

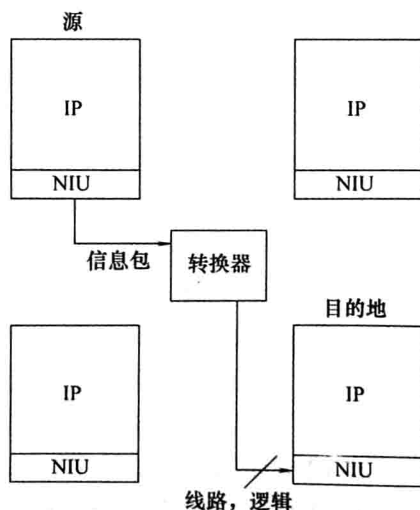


图5.24 NoC的事务层、传输层和物理层[25]

3. 更好的控制优质服务。在传输层定义的规则可以用于区分时间敏感的和尽力型的传输。信息包的优先阵列帮助实现优质服务的需求，以在严格的模块上实现实时性。

4. 灵活的吞吐量。通过分布多个物理传输链路，能够增加吞吐量，以适应系统静态或动态的需求。

5. 多时钟域操作。由于时钟的概念只针对物理层而不针对传输层和事务层，特别的，NoC 适用于包含运行在不同时钟频率下的 IP 模块的 SoC。在物理层使用合适的时钟同步电路，有单独时钟域的模块可以结合在一起，从而减少时序收敛的问题。

5.8.2 NoC 和 NIU 的实例

对于 5.7.2 节中的 Nexus 交换器，NIU 实现了 PivotPoint 系统结构，它利用 Nexus 交叉开关连接节点。图 5.25 给出了一个简单的 PivotPoint 结构。除了 Nexus 交叉开关外，FIFO 缓冲器为传送（TX）和接收（RX）通道提供了数据缓冲功能。系统信息包接口（SPI-4.2，图中简化为 SPI-4）为片对片通信实现了一个标准的协议，协议的数据速率为 9.9 ~ 16Gb/s。

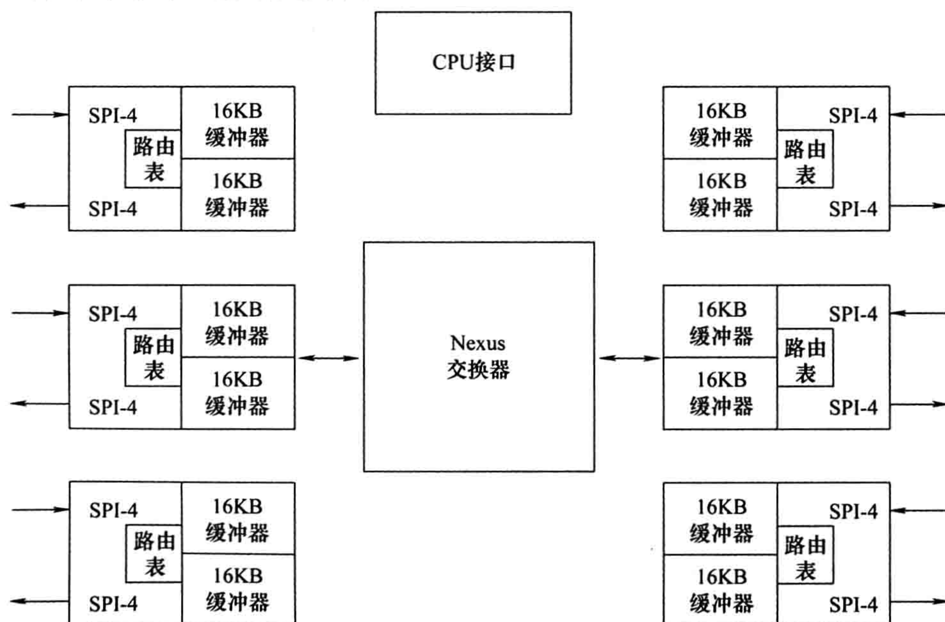


图 5.25 一个简单的 PivotPoint 结构[66]

5.8.3 总线与 NoC 比较

与总线相比，NoC 并不是没有缺点。也许 NoC 最大的一个缺点是它带来的

额外的延时。不像依靠带宽和吞吐量来决定服务质量的数据通信网络，SoC 应用常还有非常严谨的延时限制。再者，NIU 和交换结构增加了系统面积。因此，SoC 中传统网络结构的直接实现常带来不可接受的面积和延迟开销。表 5.6 给出了总线和 NoC 的对比，给出了它们在 SoC 互联质量的优点与缺点。

表 5.6 总线与 NoC 的对比[112]

总线的优缺点	NoC 的优缺点
每一个单元会附加寄生电容（－）	仅点对点单一通路线路被用于所有大小的网络（＋）
总线时序分析在深亚微米工艺上比较困难（－）	由于网络协议是全局异步的，网络线路可以被管道化（＋）
总线测试困难并且较慢（－）	内建自我检测（Buil In Self Test，BIST）快速并且完整（＋）
总线仲裁延时随着主设备的增加而增加。仲裁器要根据具体事例确定（－）	路由决策是分布式的，相同的路由器被所有大小的网络使用（＋）
带宽被附加的单元所限制并且共享（－）	聚集带宽按网络大小规模计算（＋）
一旦仲裁器被授予控制权，总线延迟为零（＋）	内部网络竞争造成一个小延迟（－）
对于小系统总线的硅开销较小（＋）	网络具有较大的硅面积（－）
任何总线几乎直接与大多数可用的 IP 核兼容，包括运行在 CPU 上的软件（＋）	面向总线的 IP 需要灵活的封装。在多处理器系统中，软件需要同步清理（－）
概念简单易懂（＋）	系统设计者需要重新学习新概念（－）

5.9 互联网络评估

已经有很多关于不同网络配置优点比较的分析[136,144,194]。下面用一个简单分析模型来评估互联网络。

5.9.1 静态网络与动态网络比较

本节很大程度上是依据 Agarwal 在网络性能上的工作，得出结论。

动态网络　假定有一个由 $k\times k$ 交换器组成的具有虫孔路由的动态间接网络。假定这个网络有 n 级，通道位宽为 w，信息长度为 l 位。在间接网络中，假定第一个网络通路地址在信息离开节点之前的一个周期中传输，因此建立互联只有 1 个周期的开销，如图 5.26所示。

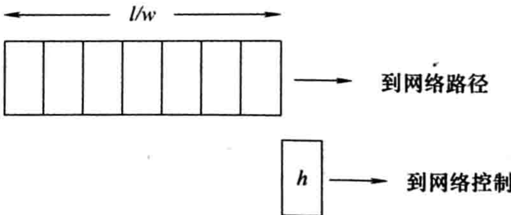


图 5.26　信息从节点到交换器的传递

假定交换器有单元延迟（T_{ch}为1个周期），在没有竞争的情况下，信息在网络中传输的总时间为

$$T_c=\left(n+\frac{l}{w}+1\right)\text{个周期}$$

在后续的分析中假定$n+l/w\gg 1$，因此有

$$T_c\approx\left(n+\frac{l}{w}\right)\text{个周期}$$

在阻塞的动态网络中，每个网络交换器都有一个缓冲器。如果一个阻塞被检测到，在这个节点就会产生一个队列；所以，N个单元每一个都需要网络的服务，单元占有率为ρ。由于在每个网络级的连接线数量是一样的，所以每个网络级所期望的占有率为ρ。在每个交换器上，信息的传输需要一个等待时间。Kruskal 和 Snir[145]已经给出了这个时间的大小（假定T_{ch}为1个周期，并且时间用周期表示）：

$$T_w=\frac{\rho(l/w)(1-1/k)}{2(1-\rho)}$$

通道占有率为

$$\rho=m\frac{l}{w}$$

式中，m为一个节点在1个通道周期中产生请求的可能性。

信息传输的总时间$T_{dynamic}$为

$$\begin{aligned}T_{dynamic}&=T_c+nT_w\\&=\left(n+\frac{l}{w}+\frac{n\rho}{2(1-\rho)}\left(\frac{l}{w}\right)(1-1/k)\right)T_{ch}\end{aligned}$$

静态网络　在静态（k，n）网络上有相似的分析。令k_d是信息传递一维所需的平均跳数。对于具有闭包的单向网络，有$k_d=\frac{(k-1)}{2}$；对于双向网络，$k_d=\frac{k}{4}$（k是偶数），信息从源单元传播到目的单元的总时间为

$$T_c=\left(h\times n\times k_d+\frac{l}{w}\right)T_{ch}$$

继续假设T_{ch}为1个周期，并且在1个基本周期中完成剩余的计算。Agarwal[8]得出的等待时间（$M/G/1$）为

$$T_w=\frac{\rho}{1-\rho}\,\frac{l}{w}\,\frac{k_d-1}{k_d^2}(1+1/n)$$

信息到达目的单元（$h=1$）的总传播时间为

$$T_{static}=T_c+nk_dT_w$$

$$= nk_{\mathrm{d}} + \frac{l}{w} + \frac{nk_{\mathrm{d}}\rho}{1-\rho}\left(\frac{l}{wk_{\mathrm{d}}}\right)(1+1/n)$$

小的 k（也就是 $k=2$，3，4）不能被用于计算过程。这里有[1]

$$T_{\mathrm{w}} = \frac{\rho}{2(1-\rho)}\ \frac{l}{w}$$

并且 $\rho = \frac{mk_{\mathrm{d}}l}{2w}$，或者对于超立方体结构为$\frac{mk_{\mathrm{d}}l}{w}$。

5.9.2 网络比较：实例

在接下来的实例中，假定 m 为0.1，m 是一个单元在任何通道周期中需求服务的可能性；$h=1$，$l=256$，$w=64$。将4×4网格（环）静态网络与 $N=16$、$k=4$、$n=2$ 的动态网络和 $N=16$、$k=2$ 平均动态网络相比较。

对于动态网络，层级的个数为

$$n = \log_2 16 = 4$$

通道占有率为

$$\rho = m\frac{l}{w} = 0.1 \times \frac{256}{64} = 0.4$$

没有竞争的信息传输时间为

$$T_{\mathrm{c}} = n + \frac{l}{w} + 1 \text{ 个周期} = \left(4 + \frac{256}{64} + 1\right)\text{个周期} = 9 \text{ 个周期}$$

等待时间为

$$T_{\mathrm{w}} = \frac{\rho(l/w)(1-1/k)}{2(1-\rho)} = \frac{0.4(256/64)(1-1/2)}{2(1-0.4)}\text{个周期} = \frac{0.8}{1.2}\text{个周期} = 0.67 \text{ 个周期}$$

因此，信息传输总时间为

$$T_{\mathrm{dynamic}} = T_{\mathrm{c}} + nT_{\mathrm{w}} = [9+4(0.67)]\text{个周期} = 11.68 \text{ 个周期}$$

对于静态网络，平均跳数是 $k_{\mathrm{d}} = \frac{\mathrm{k}}{4} = 1$，总信息传输时间为

$$T_{\mathrm{c}} = \left(h \times n \times k_{\mathrm{d}} + \frac{l}{w}\right)T_{\mathrm{ch}} = \left[1 \times 2 \times 1 + \left(\frac{256}{64}\right)\right]\text{个周期} = 6 \text{ 个周期}$$

由于

$$\rho = \frac{mk_{\mathrm{d}}l}{2w} = \frac{0.1 \times 1 \times 256}{2 \times 64}\text{个周期} = 0.2 \text{ 个周期}$$

对于低 k，T_{w} 为

$$T_{\mathrm{w}} = \frac{\rho}{2(1-\rho)}\ \frac{l}{w} = \left[\frac{0.2}{2(1-0.2)} \times \frac{256}{64}\right]\text{个周期} = 0.5 \text{ 个周期}$$

等待时间为

$$
\begin{aligned}
T_{\text{static}} &= T_{\text{c}} + nk_{\text{d}}T_{\text{w}} \\
&= (6 + 2 \times 0.1 \times 0.5)\text{个周期} \\
&= 7\ \text{个周期}
\end{aligned}
$$

5.10 总结

互联子系统是SoC的主干。系统的性能会被互联的限制所压制。由于它的重要性，大量的工作是优化开销-性能的互联策略的。

除了完全定制的设计，SoC有两种差异明显的互联方法：基于总线的和NoC的。然而，这些都是可实现的方法。NoC可以连接这样的节点，它们自己是基于总线的处理器或其他IP的集群。

过去大多数SoC主要是基于总线的。被连接的节点数量是很少的（也许是4个或8个IP），并且每个节点仅由一个IP组成。它保持着一个被测试过的互联方法，这个方法能够简单使用。协议标准和总线封装的使用使得IP核集成的任务具有更少的错误风险。而且，大量总线选择允许使用者在复杂度、使用方便性、性能和通用性方面做出权衡。

随着互联节点数量的增加，基于总线方法的带宽限制变得更加明显。交换器解决了带宽的限制，但是增加了额外的开销。开销取决于配置和额外的延迟。由于交换器（无论静态或者动态）被集成到IP中，能够被很好地支持，并且有处理突发事件的工具。它们将成为SoC互联的标准，特别是对于高性能系统。

对基于总线或交换的互联进行性能建模是SoC设计中很重要的一部分。如果一开始分析到基于总线的互联有低效带宽和系统性能，就可以选择基于交换的设计。初始分析和设计选择通常是基于分析模型进行的，但是一旦只有很少的选择，一个更加全面的仿真需要被用于确定最后的选择。SoC的性能取决于互联设计的配置和承载量。

在NoC实现中，网络接口单元充当一个重要角色。它具有相对较小的开销，实现了互联的分层。它允许设计的重新构造，并支持加入新的交换器进行扩展，而不影响SoC的上层实现。NoC使用的增长促进了SoC的发展。

在SoC互联中有很多话题不在本章的讨论范围，如片上通信协议的设计与验证的结合[46]、自测时间信息包的交换[105]、功能模块化、NoC系统的验证[210]和AMBA 4技术在重配置逻辑上的优化。本章的内容和其他相关内容，如Pasrichahe和Dutt[193]，这里只给出了基本的介绍，读者可以根据这些内容，为SoC互联的深入发展做出贡献。

5.11 习题

1. 一个独占分离（地址加上双向数据总线）总线是32位 + 64位位宽的。一个典型的总线事务（读或写）使用32位内存地址，紧接着有一个128位数据传输。假设访存时间是12个周期。

（a）画出读和写的时序图（假设没有竞争）。

（b）一个事务的（数据）总线占有率是多少？

2. 如果四个处理器使用上题中描述的总线，理想情况下（没有竞争）每个处理器每20个周期产生一个事务。

（a）给定的总线占有率为多少？

（b）使用没有重新提交的总线模型，实际的占有率为多少？

（c）使用有重新提交的总线模型，实际的占有率为多少？

（d）（b）和（c）的结果对系统性能的影响是什么？

3. 查询目前使用AMBA总线的产品的资料；找到至少三个不同的系统，列表显示它们各自的参数（AHB和APB）：总线位宽、带宽和每条总线的IP使用者数量的最大值。尽可能给出详细的信息。

4. 查询使用CoreConnect总线的当前产品的资料，找到至少三个不同的系统，列表显示它们各自的参数（PLB和OPB）：总线位宽、带宽和每条总线的IP使用者数量的最大值。尽可能给出详细的信息。

5. 讨论在将总线封装从AMBA总线转换为CoreConnect总线时会遇到的问题。

6. 一个静态交换互联是4×4环面的（二维），具有虫孔路由。每条通路是双向的，有32条线路；每条线路的速率为400Mb/s。对于由8位头部和128位“负载”组成的信息，那么

（a）信息从一个节点传输到相邻节点的期望延迟（用周期数表示）是多少？

（b）节点之间的平均距离和平均信息延迟是多少（用周期数表示）？

（c）如果网络的占有率为0.4，由于信息阻塞（等待）造成的延时是多少？

（d）信息传输总时间是多少？

7. 一个动态交换互联用于连接16个节点，它使用一个2×2交叉开关的交换网络。在一个2×2开关上传输需要一个周期。每条通路是32线路双向通信的；每条线路的速率是400Mb/s。对于由8位头部和128位“负载”组成的信息，那么

（a）一条信息从一个节点传输到另一个节点的期望时间是多少（用周期数表示）？

（b）画出这个网络。

（c）如果网络的占有率为0.4，信息等待时间是多少？

（d）信息传输的总时间是多少？

8. 交换网络的二分带宽，是将网络分为两个相同部分（节点数）后，通过线路的最大的可用带宽。按上述描述，静态网络和动态网络的二分带宽是多少？

9. 查询资料，找到至少三个不同的 NoC 系统；比较它们底层的交换器（找到至少一个动态和一个静态的例子）。用表格显示详细信息。

第6章　定制与可配置性

6.1　引言

为了扩大SoC的适用范围同时降低成本，可以用一般的可定制的硬件平台提高特定应用程序效率。本章主要讲述一些定制技术，尤其是基于可配置性的定制技术。这里所说的可配置性包括一次可配置性和可重复配置。前者表示面向应用程序的芯片定制在制造之前或之后只进行一次；后者表示芯片的定制在制造后还可以多次进行定制。

在设计阶段的定制，尤其是那些在设备制造过程中进行的定制，通常性能会很高，但在设计开展之后会牺牲一定的灵活性。这种后制造的灵活性是通过具有不同程度可编程性的器件来实现的，包括粗粒度可重构结构（Coarse-Grained Reconfigurable Architecture，CGRA），专用指令集处理器（Application-Specific Instruction Processors，ASIP），细粒度FPGA，DSP，GPP。可编程性与性能之间的关系如图6.1所示，本书第1章已经介绍过。

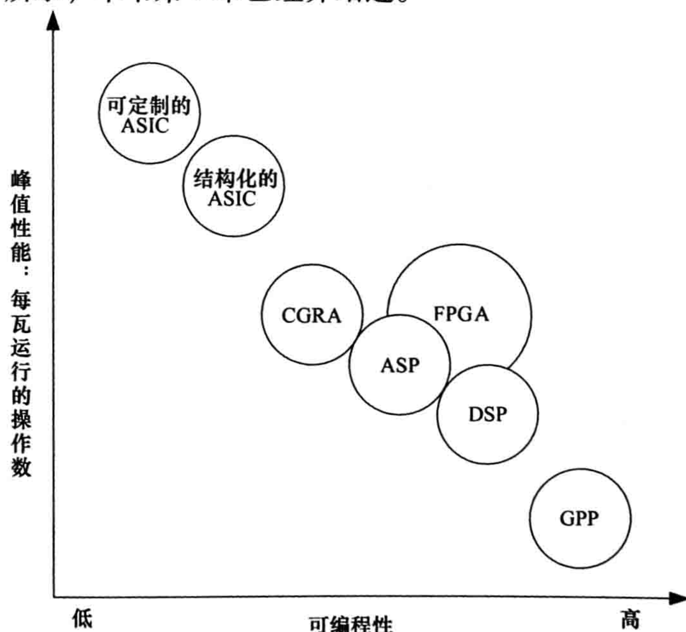


图6.1　可编程性与性能之间的关系

与可定制的 ASIC 技术相比，结构化的 ASIC 技术支持设计之前的有限的可定制性。在图 6.1 中，ASIP 被假定为在制造时定制。当然，如果 ASIP 在 FPGA 中作为软件处理器使用，那么 ASIP 可以在编译时刻定制。这种可定制的 ASIP 处理器将会在 6.8 节讲述。

有很多定制 SoC 的方法，本章着重讲述以下三种：

1. 指令处理器的定制（见 6.4 节和 6.8 节），揭示了（a）处理器族的可用性和（b）特定应用程序处理器的生产是怎样提供结构优化的，如 VLIW、矢量化、融合操作（fused operation），以及多线程等优化方式，最终满足性能、面积、能源效率及价格的需求。

2. 可重构架构的定制（见 6.5 和 6.6 节），讲述细粒度可重构功能单元（Functional Unit，FU）与相关的互联资源是很通用的，但会导致很大的开销，因此可以用较多的粗粒度的模块来减少这种开销。

3. 用于实现优化的定制技术，如特定实例设计（见 6.7 节）和运行时重构策略（见 6.9 节），以及在性能、尺寸、功率、资源利用率之间权衡的评估方法。

其他定制技术不会做详细的介绍，如基于多处理器的技术。这些相关的技术见 6.10 节。

6.2 估算定制的有效性

在设计的过程中，对不同的定制技术的有效性进行估算与比较很重要。评估的方法很简单。对于给定的度量标准，如延迟、面积、功耗，假设对于设计的 β（百分比）部分 α 为其改进系数。那么这个度量标准被提高了，即

$$(1-\beta)+\alpha\times\beta$$

这种度量方式让人联想到 G. Amdahl 对并行处理器的分析。例如，在著名的 90：10 法则中，一个好的提高速度的方法是让 10% 的代码占据了 90% 的执行时间。如果这 10% 的代码执行速度提高 k 倍会怎么样呢？

在上面的表达式中，$\alpha=1/k$，$\beta=0.9$，得到 $(k+9)/(10k)$。假设 $k=10$，那么这个执行时间减少为原来的 19%，加速比为 5.26。

但是，如果这些可加速的代码的执行时间的所占的比例从原来的 90% 降到 60%，会发现加速比仅为 2.17 倍，几乎变成原来的 1/2。

注意，提供用于有效性估计的代码量不影响上面的结果；而是可定制部分的代码所执行时间占总时间的比例影响上面的结果。

这种方法可用于很多方面，如考虑用嵌入式粗粒度模块定制细粒度可重构架构（后面将会介绍）。假设有 50% 的细粒度结构可以被粗粒度块替换，这部分速度会提高三倍，面积使用率会提高 35 倍。可以发现，设计比以前快了 50%，总

面积几乎减少了1/2。

然而，有一些限制因素影响了可定制性的实施。实现可定制处理器的一些工具不能像用于实现非定制性处理器那样成熟，如性能分析器和优化器；另外，向后兼容与验证也会有困难。一种解决方案是开发与现存的非定制模块相兼容的可定制处理器，这样在可定制与非可定制技术[93]之间就存在了互用性。

6.3 SoC 定制综述

定制是一个为了满足应用需求和实施约束条件而对设计进行优化的过程。该过程可以在设计或运行时刻实现。设计由两个阶段组成：制造阶段与编译阶段。在制造阶段生产出来的设备，如果是可配置的，那么制造阶段完成后，它可以由编译或运行时产生的程序实例化。

通常有三种方式执行运算：标准的指令处理器、ASIC 及可重构的设备。

1. 标准的指令处理器，像 ARM、AMD 及 Intel 指令处理器，在定制的制造阶段产生支持固定的指令集架构的设备，而在编译阶段为该架构产生指令，在运行时定制是指定位或产生用于运行时的合适代码。

2. 对于 ASIC，许多实现应用程序功能的定制是在制造阶段进行的。因此定制的设备具有很高的性能，但灵活性差。这是因为如果在制造前若没有设计一些可扩展的功能，那么新功能将很难添加进去。结构化的 ASIC，像门阵列或者是标准单元技术，通过限制用户对芯片定制的选择（如对金属层的选择）减少了设计工作量。可一次定制的反熔丝技术可以用于这个领域。

3. 可重构设备主要包括 FPGA、复杂可编程逻辑器件（Complex Programmable Logic Device，CPLD)，以及含有支持定制指令的可重构架构的指令处理器。在定制的制造阶段会产生含有可重构架构的设备，这种设备含有可重构元素，元素之间通过可重构的互联资源进行连接。在编译时刻，配置信息在设计的描述中产生，在运行时使用这些配置信息定制设备。

标准指令处理器以通用为目的。但是，也可以定制用于特定应用程序的指令集和架构。例如，标准的指令集可以通过去掉一些不用的指令或者是添加一些新的指令来提高性能。指令处理器的定制，可在 ASIC 技术的制造阶段或者在可重构硬件技术的配置阶段完成。

- 制造阶段定制的处理器主要包括美国 ARC 公司和美国 Tensilica 公司的处理器。通常一些构件在制造阶段是硬连线的，这是为了支持特定域的优化，如为特定应用程序定制指令等。在 ASIC 技术中，为了减少风险，通常在设计实施之前要有可重构的原型。
- 对于美国 Xilinx 公司的 MicroBlaze[259] 或美国 Altera 公司的 Nios[11] 等软核

处理器，挑战是如何有效地支持指令处理器。这些处理器使用像 FPGA 这样的具有可重构架构的资源。可以通过探索专用设备特点或者运行时可配置性[213]，来提高可编程器件的使用效率。

- 另一个选择是用 ASIC 技术实现指令处理器。这需要开发一个合适的接口，以使处理器能够使用通过可重构架构实现的定制指令。例如 Stretch[25] 公司的软件可配置处理器，它由美国 Tensilica 公司的 Xtensa 指令处理器及粗粒度可重复配置架构组成。在应用程序需要与它们的体系结构相匹配的资源时，这些设备可以提供比 FPGA 更高的效率。

考虑使用哪种技术关键得看处理器的数量。回忆本书第 1 章的内容，产品成本包括一个与处理器数量无关的固定成本，还包括因处理器数量不同而可变的成本。像 PFGA 这样的重构技术只有很少的固定成本，但比起 ASIC 技术，它有更高的可变成本。因此，在处理器的数量方面低于某个阈值时，重构技术的成本最低。并且，重构技术的这个阈值会随着时间的推移而产生波动，因为封装成本和其他固定成本会随着新技术的产生迅速增加。

有很多对可定制 SoC 进行分类的方法。一个可定制的 SoC 通常包括一个或多个处理器、可重构逻辑及存储部件。可以根据可重构逻辑和这种处理器的连接方式对这种 SoC 进行分类[61]。

图 6.2a 给出了连接到系统总线的可重构架构。图 6.2b 给出了可重构架构作为 CPU 的一个协处理器的情况，比起图 6.2a 所示架构它与处理器的连接更近一些。

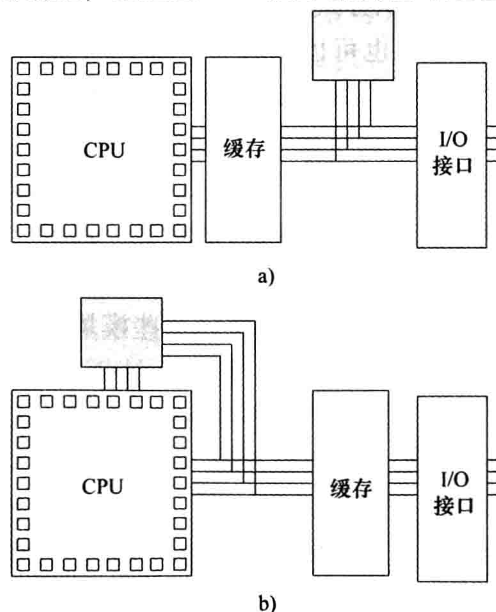


图 6.2　四种定制 SoC[61,244]（阴影方块代表可重构逻辑）
a）附加出单元　b）协处理器

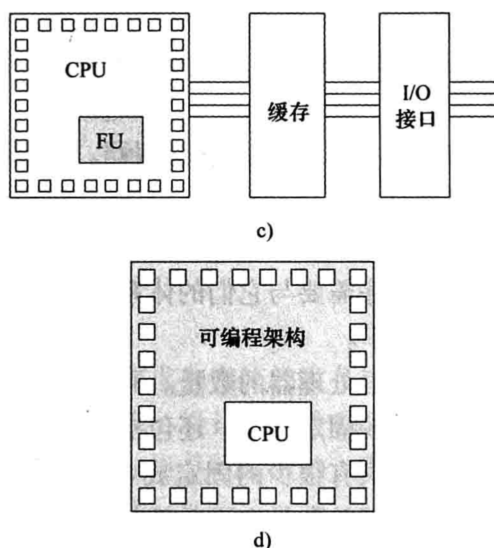


图 6.2 四种定制 SoC[61,244]（阴影方块代表可重构逻辑）（续）
c）可重构 FU d）嵌入可重构逻辑的处理器

图 6.2c 给出了一个处理器与这种架构结合更紧密的体系结构。这种情况下，可重构逻辑作为处理器的一部分，可能会形成一个可重构的子单元来支持定制指令。美国 Stretch 公司的软件可配置处理器就是这样组织的。

图 6.2d 给出了另一种组织方式。在这种方式下，处理器内嵌在可编程架构中。处理器可以是硬核[261]，也可以是软核，通过可重构逻辑本身的资源来决定。这样的实例包括先前提到的 MicroBlaze 和 Nios 处理器。

也可以将可配置的模拟电路与数字电路的功能整合在同一个芯片上。例如美国 Cypress 公司的可编程片上系统（Programmable SoC，PSoC）设备[68]有一个模拟模块阵列，它可以配制成多种部件的组合，包括比较器、过滤器、具有可编程互联结构的模-数转换器等。这些组件及可重构数字模块的 I/O 可以很容易地连接到 I/O 出端口。并且在运行时，这些模块可以进行重构以执行不同的功能。

6.4 定制指令处理器

桌面计算机的微处理器为通用计算而设计。SoC 上的指令处理器通常是为特定类型的计算而设计，如媒体处理或数据加密。因此可以通过定制得到非常符合特定意图的处理器，并且消除不需要的硬件部分。尽管很多技术可用于软核处理器的设计，但这种定制通常在制造之前进行。这种定制允许设计者在提高产品多

样性的同时，优化它们的设计以满足一些需求，如速度、面积、功耗和准确性等。

6.4.1　处理器定制方法

处理器的定制方法主要有两种。第一种方式是通过特定应用程序域的处理器族产生。例如，英国 ARM 公司提供的一些处理器，像 Cortex-A 系列用于支持需要大量计算的应用程序，Cortex-R 系列用于实时处理，Cortex-M 系列可用作嵌入式应用程序微控制器（Cortex-M1 处理器实现了 FPGA 的优化），SecurCore 系列处理器用于防篡改智能卡。每一系列包含一类不同特征的处理器，允许设计者根据自己在功能或性能及功耗等的需求，选择合适的处理器。

第二种方法是提供能生成定制的处理器的工具。美国 ARC 公司和美国 Tensilica 公司生产了一些设计工具，允许 SoC 设计师使用图形化用户接口或者基于体系结构描述语言来配置和扩展一个处理器。使用这些工具，设计师可以通过对现有的处理器进行修改，如添加一些需要的特征或删除不需要的特征，从而生产目标处理器。此外，设计师还可以通过添加定制指令来扩展内核的体系结构，从而使处理器对最终应用程序进一步优化。

为了对尺寸、功耗、应用程序性能进行优化设计，一些 SoC 设计工具还提供最终的芯片面积和内存需求的说明。设计者可以配置与处理器内核相关的一些特征，如缓存的类型与大小、DSP 子系统、计时器及调试组件；还可以配置处理器内核的一些内部特征，如内核中寄存器的类型与大小、地址宽度、指令集选择等。其最终目的是对性能与面积之间的权衡提供一种优化方案。SoC 工具通常包括以下一些具体功能：

- 集成不同来源的 IP 核。
- 产生系统验证的仿真脚本与测试实例。
- 扩展软件开发工具的功能，如支持定制处理器的定制指令。
- 自动产生 FPGA 设计的仿真平台。
- 在被授权的芯片规范中选择配置的文档，以及用户级文档。

这些工具通常可以在几分钟内配置和产生一个定制的处理器。并且通过产生所需要配置的具体信息，如测试时所需要的信息、下游开发工具及开发文档的信息，可以让配置程序减少流片时间，降低项目风险。

有关定制嵌入式处理器的开发工具及其应用程序的更多的信息，都可以在不同的文献资料中找到[127,207]。

6.4.2　架构描述

现代处理器变得越来越复杂，因此，在设计处理器和与处理器相关的软件工

具时，提供一定程度的自动化很有必要。没有自动化，对每一个应用，设计和验证工作需要从头构建一个处理器及其工具。

体系结构描述语言或处理器描述语言对于处理器及其工具的自动化发展起到很大的作用[179]。对于设计者来说，这些语言在应用于不同的应用程序时应该很简洁、高效；并且用于描述真实、有效的设计时容易理解。已经有很多语言可用来描述指令处理器。这些语言的目标是捕捉处理器的设计以便使处理器和辅助工具可以自动产生。

体系结构描述语言可以通过描述风格、描述层次及提供的自动化的类型进行分类。例如，可以把体系结构描述语言分成行为级、结构级，或者两者之间的结合。

行为级描述以指令集为中心：设计者确定指令集，然后使用工具产生编译器、汇编器、连接器及仿真器。nML[87]和 TIE[240]语言属于行为级描述语言的范畴。对于代码生成工具，如 GURG[100]，当指令可以通过语法树描述时，很容易自动产生用行为级描述语言实现的编译器后端。许多行为级描述语言支持处理器的硬件综合，综合工具可用于上述的实例。在 TIE 语言中，综合被简化了，因为所基于的处理器是固定的，只有扩展部分可以由设计者指定。nML “Go” 工具可由指令集自动设计处理器结构[86]，通过显示的资源共享，如寄存器堆，推断出处理器结构。

行为级描述的主要优势在于其高度的抽象性：生成一个定制的处理器，只需要一个指令集规范。它的主要缺点是，硬件实现时缺乏灵活性。综合工具必须保证微体系结构[87]甚至是整个基础芯片[240]的一些特征不变。当考虑到资源共享[50]时，对自动化设计来说，仅依靠指令集实现有效的综合是很困难的。

结构级描述可以获取处理器内部的功能单元、存储资源及互联资源。PREE[269]是一个建立在 C++ 之上的库，它可以通过结构描述产生 FPGA 软件处理器。设计者可以删除一些指令或者是改变功能单元的实现方式，因为 SPREE 提供一种方法连接功能模块，并内部支持常见功能，如旁路网络和互锁等。

结构级描述的主要优势是它可以直接转化成一种适于综合成硬件的形式。此外，大多数结构级描述会保持硬件描述语言（Hardware Description Language，HDL）的原则。这种原则有利于不同的微体系结构的描述，如超标量和多线程等。结构级描述的主要缺点是抽象层次较低：设计者需要手动地指定功能单元和控制结构。

表 6.1 给出了一些体系结构描述语言的特征。对于每种语言，都有它的描述风格，描述范围（整个处理器或者仅是指令集）及自动产生处理机系统的工具。

表 6.1　一些体系结构描述语言的特征

特　征	Expression[114]	CUSTARD[77]	LISA[118]	SPREE[269]	nML[239]	TIE[240]
整个处理器	√	√	√	√	√	
工具链配置	√	√	√	√	√	√
产生硬件		√	√	√	√	√
存储系统	√	√	√			
行为级	√	√	√		√	√
结构级	√		√	√		

一些系统，如 LISA[119]，既含有结构级信息又含有行为级信息。这种系统结合了行为级描述和结构级描述的优点，但是需要保证相关行为及结构元素的一致性。

除了 TIE 之外的所有语言都可以用于描述整个处理器，也就是说可以对整个处理器进行设计，而不仅是指令集。然而，许多处理器描述语言专为一个基本的处理器架构而设计，如在 LISA 及 nML 中的顺序执行。6.8 节将介绍的可定制的线程结构（Customizable Threaded Architecture，CUSTARD）[77]，是基于 MIPS 指令集，可以同时支持各种定制选项，如多线程的类型和定制指令的使用等。

这些语言需要使用以下工具[179]：

- 模型生成器。这类工具可以生成硬件原型和验证模型，来检查体系结构描述是否符合需求。
- 测试生成器。这类工具用于产生测试程序、断言和测试用例。
- 工具包生成器。这类工具用于分析、检索、编译、仿真、装配和调试设计。

6.4.3　自动识别定制指令

有多种方法可以通过对高级应用程序的描述自动识别扩展指令集。可以试探性地将一些数据流图（Data Flow Graph，DFG）节点聚集成串行或并行模板。通过在子图中加一些 I/O 约束来减少搜索空间的指数增长。

各种各样的体系结构优化方案（一些已经在前面的章节中讲过）有助于自动化设计，例如[110]：

- VLIW 技术使得一条指令可以支持多种独立的操作。VLIW 格式将一个指令分成多个字段，每个字段代表一个操作。如果指令集在设计时使用 VLIW 技术，那么源语言编译器可以使用软件流水和指令调度技术将多个操作打包成一个单独的 VLIW 指令。
- 建立对多个数据元素进行操作的矢量运算可以增加吞吐量。矢量运算的

特征由其在每个数据元素上的操作和在并行运算中数据元素的数量（即矢量的长度）决定。例如，一个宽度为四的定点加法操作包括两个输入矢量，每个都包含四个整数，并产生一个结果为四个整数的矢量。

- 将多个简单操作组成一个混合操作。一个混合操作可能有一个或多个输入操作数是固定的常数。使用混合操作代替简单操作可以减少代码量、控制带宽，并可能减少对寄存器堆端口数量的要求。并且，混合操作的延迟可能会低于合并前的简单操作的延迟之和。

约束传播技术的应用会产生有效的计数算法。然而这种方法的使用性仅限于大约 100 个节点的数据流图。通过在子图中添加附加约束，如只允许单个输出或者是连通性约束，搜索空间可以显著降低。

识别输入和输出约束下的扩展指令集可以归为一个整数线性规划问题。Biswas 等人[45]基于 I/O 约束提出了一种对 Kernighan - Lin 启发式规则的扩展方法。优化往往限于一个近似搜索算法或者一些人工约束（如 I/O 约束）以使对子图的枚举容易处理。

整数线性规划可以用实际的数据带宽约束及数据传输代价代替 I/O 约束。因为产生的扩展指令集可能会有无限个输入和输出。将数据带宽信息直接整合到优化进程中，或者考虑数据在核心寄存器堆与定制状态寄存器[28]之间的传输代价，可以获得满意的结果。含有结构可见的状态寄存器的基线机（baseline machine）可实现这种方法。

定制指令处理器时有很多方法，如技术感知方法。这种方法包括一个集群策略，可用于评估基于 FPGA 的具体的定制指令查找表（Lookup Table，LUT）的资源使用情况，而无需经过整个综合过程[148]。再如应用程序感知方法，在视频应用程序中时，可以利用合适的中间表示和循环的并行性[165]。还有转化感知方法可以采用基于组合但逐步搜索源代码转化设计空间和指令集扩展设计空间的方法[182]。

6.5　重构技术

在各种重构技术中，FPGA 是最常见的。为了支持高性能设计，它的容量和性能在过去的几年里得到了迅速的改善。FPGA 低成本和支持快速开发的特点使得它成为了一些设计的理想平台，能满足对快速上市有需求的项目，如教育和学生项目。

下面介绍支持 FPGA 及其他可重构设备的可重构逻辑。可重构逻辑由一组可重构的功能单元、可重构互联资源，以及一个连接系统其他逻辑的灵活接口组成。下面将介绍每个组件，并展示它们在工业及学术上的可重构系统中的使用方

法。这里使用的方法是Todman等人[244]提出的。

对逻辑中的每个组件，都要在性能与灵活性之间进行权衡。一个灵活性高的架构通常较大，并且比灵活性低的架构慢。但灵活性较高的结构能更好地适应应用程序的需求。这种灵活性与性能之间的权衡会影响可重构系统的设计。表6.2给出了可重构结构和设备的比较。由于篇幅有限，一些相关的设备（如英国Elixent公司[82]的设备）没有包含在内。

表6.2 可重构结构和设备的比较

结构或设备	粒度	基本的逻辑组件	路由结构	嵌入内存	特点
Actel Axcelerator[2]	细	四输入复用器与反相器	横向和纵向路由	4Kbit块	反熔丝，低功耗模式
Actel IGLOO[3]，ProASIC[4,5]	细	三输入逻辑功能	横向和纵向路由	4Kbit块或1K闪存	基于闪存，低功耗模式
Altera Stratix Ⅲ[15]，Stratix Ⅳ[16]	细	八输入自适应逻辑模块	横向和纵向路由	640bit、9Kbit、144Kbit块	DSP模块，1.6～8.5Gb/s I/O，可编程能力
Xilinx Virtex Ⅱ Pro[262]，Virtex 4[263]	细	四输入显示查找表	横向和纵向路由	18Kbit块	乘法器模块，DSP块，PowerPC 405处理器，3.1～6.5Gb/s I/O
Xilinx Virtex 5[264]，Virtex 6[265]	细	六输入或者两个五输入查找表	横向和纵向路由	36Kbit块	DSP，6.5Gb/s I/O PowerPC 440处理器
Lattice XP2[149]	细	四输入查找表	横向和纵向路由	18Kbit块	DSP，内置闪存
SiliconBlue iCE65[219]	细	四输入查找表	横向和纵向路由	4Kbit块	低功耗模式，内置闪存
Silicon Hive Avispa[220]	粗	ALU，移位器，累加器，乘法器	总线	4×4K双端口本地数据存储	75个功能单元，95个寄存器的堆，OFDM无线应用程序

注：OFDM，Orthogonal Frequency-Division Multiplexing，正交频分复用。

6.5.1 可重构的功能单元

可重构FU可以是粗粒度的，也可以是细粒度的。细粒度FU通常可以用一位或几位实现一个功能。常见的细粒度FU都是些小的LUT，可用于实现大多数工业FPGA中的逻辑。而粗粒度的FU，通常比较大，可能由算术单元和逻辑单元（Arithmetic and Logic Unit，ALU）甚至是较大的存储空间组成。本节将更详细地描述这两种功能单元。

许多可重构系统使用工业 FPGA 作为可重构逻辑载体。这些工业 FPGA 包含三～六个输入 LUT，每个 LUT 都可以看成一个细粒度的 FU。图 6.3a 给出了一个三输入 LUT。通过移位，该功能单元可以实现任何一种多达三输入的功能。显然，LUT 的扩展需要更多的输入。通常 LUT 要结合成簇，如图 6.3b 所示。图 6.4 给出了商业体系结构的逻辑块，是在两个流行的 FPGA 系列中的簇。图 6.4a 所示的 Altera LAB 是 Altera Stratix 设备的一个簇；美国 Altera 公司把这些簇称为逻辑块阵列（Logic Array Block，LAB）[14]。图 6.4b 所示的 Xilinx CLB 是 Xilinx 结构[262]的一个簇；Xilinx 把它们称为“可配置逻辑块”（Configurable Logic Block，CLB）。

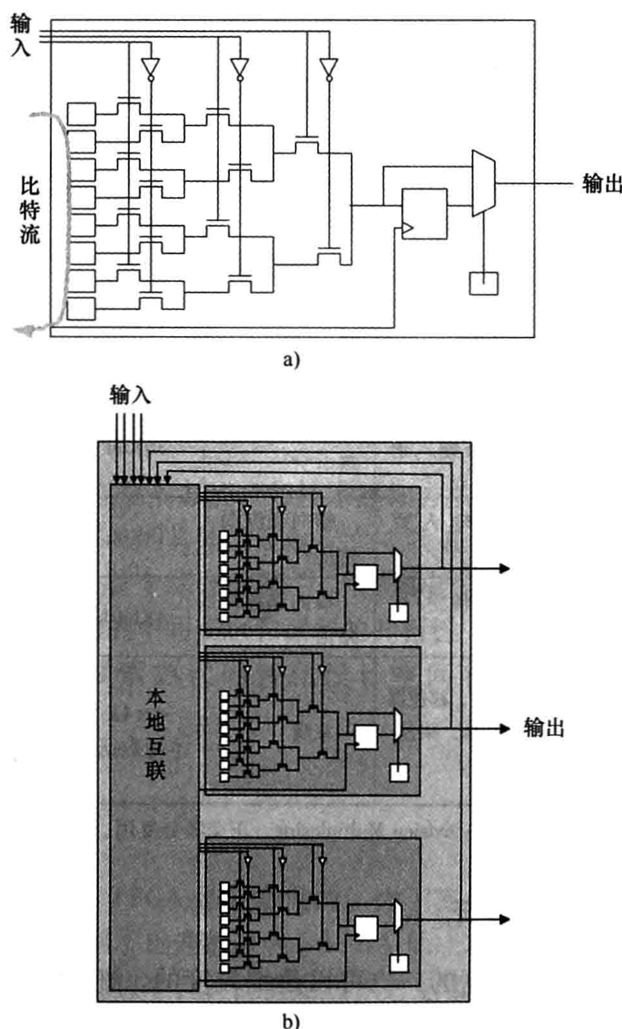


图 6.3　细粒度可重构 FU[244]

a）三输入 LUT　b）LUT 族

在图 6.4a 中，每个标为“LE”的块只表示一个 LUT；而在图 6.4b 中，每个标为“片（Slice）”的块表示两个 LUT。

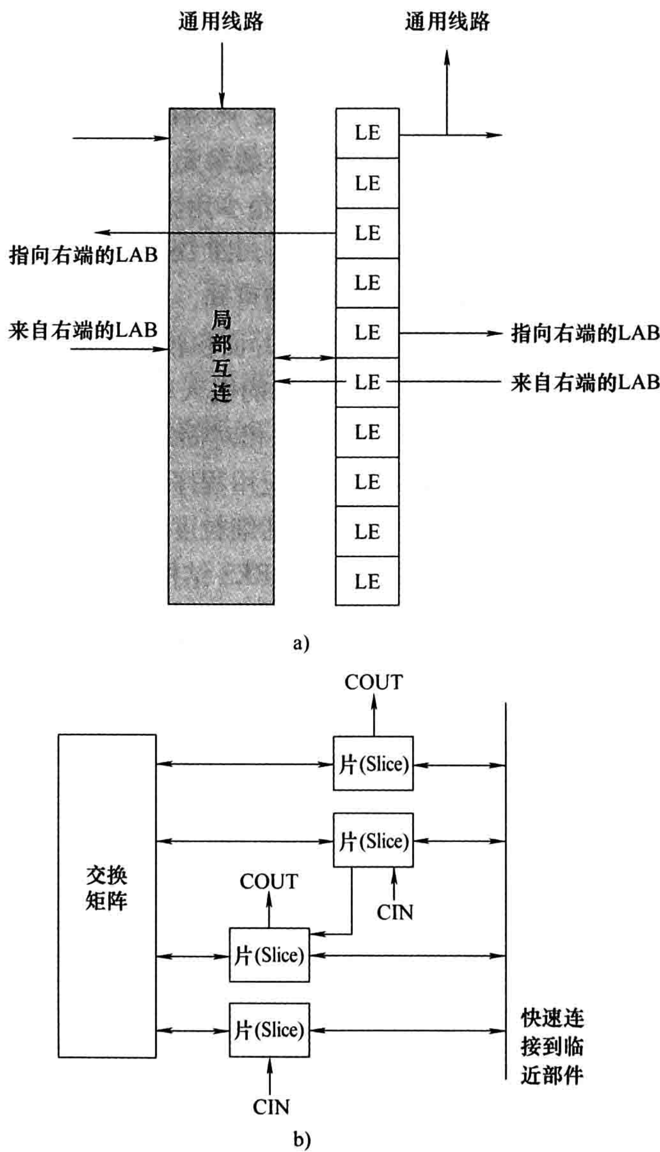


图 6.4　商业体系结构的逻辑块

a）Altera LAB[14]　b）Xilinx CLB[262]

包含 LUT 的可重构逻辑是灵活的，它可以用来实现任何数字电路。然而，与下面要描述的粗粒度架构相比，这些细粒度的架构面积较大、延迟较长、功耗较高。考虑到这些结构通常用于计算，FPGA 厂商添加了一些额外的特征，如进位链和传递链，从而可以减少实现常见的算术与逻辑运算时的开销。

虽然通过添加对常见功能的架构支持，工业 FPGA 的效率得到了提高，甚至可以更高、嵌入得更大，但这些 FU 的灵活性与可重构性较差。这里有两种包含粗粒度 FU 的设备。

第一，通过加入粗粒度模块，许多主要由细粒度 FU 组成的商业 FPGA，变得日益强大。例如，早期的 Xilinx Virtes 器件包含 18×18 位的嵌入式乘法单元[262]。在实现需要大量乘法运算的算法时，这些嵌入的单元可显著改善密度、速度和功耗。但是对不需要执行乘法的算法，这些模块很少用到。美国 Altera 公司的 Stratix 系列设备包含更大、更灵活的嵌入式粗粒度块，即 DSP 块，它可以实现加法和乘法操作。通过对比这两个器件的灵活性与开销可知，美国 Altera 公司的 DSP 块比美国 Xilinx 公司的乘法器更灵活，但占用的芯片面积较大，并且运行乘法任务时较慢。最新的美国 Xilinx 公司的设备包含更复杂的嵌入单元，叫 DSP48。应该注意，这些嵌入块虽然能消除内部的可重构互联，但相对固定的位置会导致线路拥堵和线路开销。而且，对于不使用这些嵌入块的应用程序来说也会产生开销。

第二，虽然上面描述的商用处理器既包括细粒度块又包括粗粒度块，但还有一些器件只包含粗粒度块。粗粒度结构如 ADRES 结构，如图 6.5[171] 所示。在这种器件中，每个可重构 FU 包含一个 32 位的算术逻辑单元，通过重构可以用两个小的寄存器堆实现加法、乘法及逻辑运算中的任何一种运算。显然，这种 FU

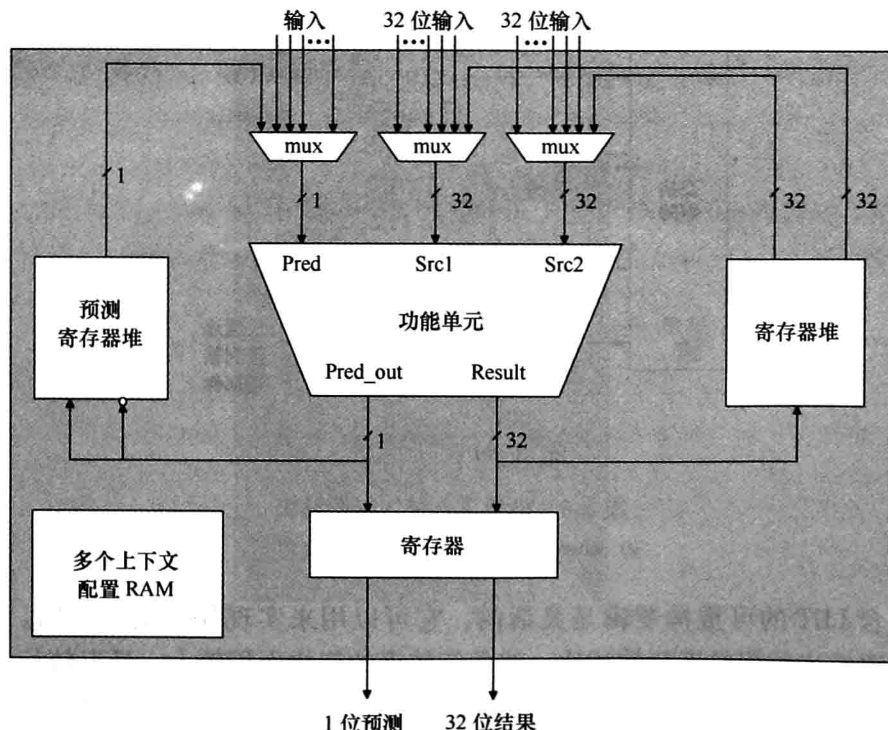


图 6.5　ADRES 可重构 FU[171]（Pred 是一位控制位，它控制功能单元选择 Src1 还是 Src2）

没有前面描述的细粒度 FU 灵活；但是，如果应用程序需要的功能与 ALU 的功能相匹配，在这种体系结构中，这些功能可以高效地实现。

6.5.2　重构互联

不管一个器件含有细粒度 FU 还是粗粒度 FU，或者是两者的结合，FU 之间的连接都要很灵活。因此，在体系结构中需要考虑互联的灵活性与速度、面积、功耗之间的权衡问题。

可重构互联体系结构可分为粗粒度结构和细粒度结构。这两种结构是基于线路间切换的粒度进行划分的。图 6.6 所示的结构，是两个总线之间的一种灵活的互联方式。在图 6.6a 所示的细粒度结构中，可以独立地切换每根线；而在图 6.6b 中，整个总线作为一个单元进行切换。图 6.6a 所示的细粒度路由结构更灵活一些，因为每位不必都按相同的方式路由；图 6.6b 所示的粗粒度结构包含的编程位较少，但功耗较低。

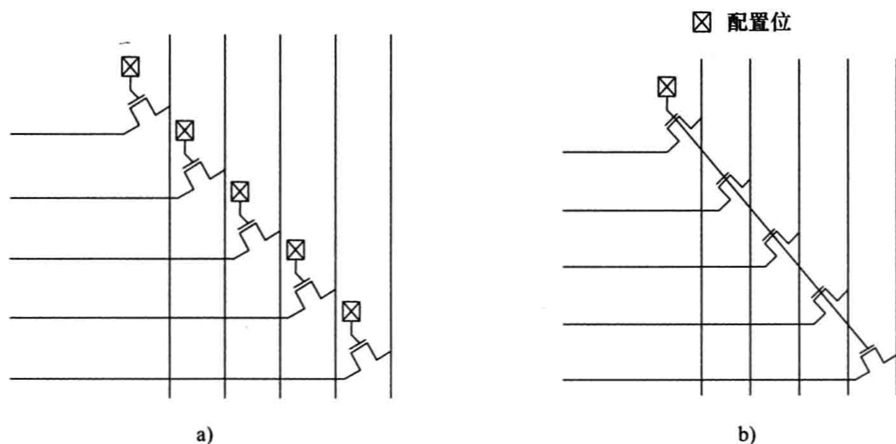


图 6.6　路由结构
a）细粒度　b）粗粒度

细粒度路由结构通常用于工业 FPGA 设备中。在这些设备中，多个 FU 通常组织成网格模式，并且使用水平和垂直通道进行连接。已经有很多关于互联的拓扑结构优化问题的研究[157]。

粗粒度路由结构通常用在包含粗粒度功能单元的设备中。图 6.7 给出了两个粗粒度路由结构的示例。图 6.7a 所示的路由结构在 Totem 可重构系统[60]中使用；这种路由结构设计灵活，可以在 FU 之间提供任意的连接方式。图 6.7b 所示的路由结构用于 Silicon Hive 可重构系统，其灵活性较差，但速度快、面积小[220]。在 Silicon Hive 结构中，只有具有通信可能的单元之间才有连接。

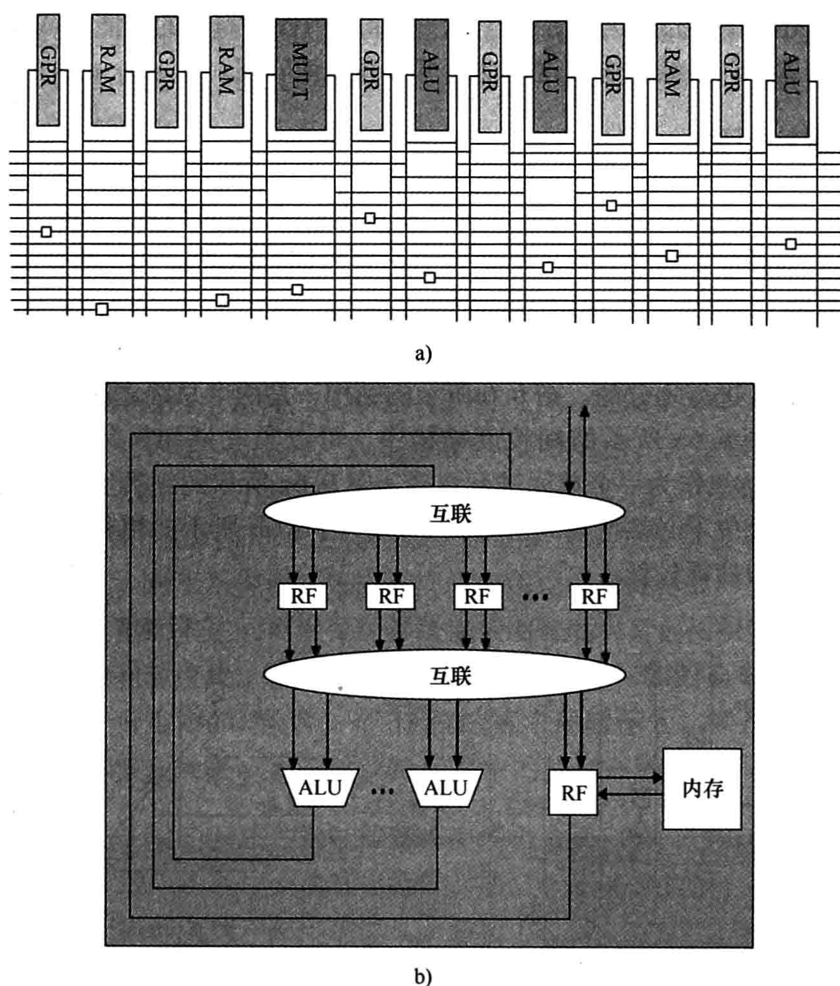


图 6.7 粗粒度路由结构的示例

a）Totem 粗粒度路由结构[60] b）Silicon Hive 粗粒度路由结构[220]

6.5.3 软件可配置处理器

软件可配置处理器是由美国 Stretch 公司引入的。它们有一种体系结构可以将传统的指令处理器与可重构架构相结合，从而允许应用程序动态地定制指令集。这种体系结构有两种优点：第一，通过开发数据并行性、操作专业化及更深的流水线，可获得较高的性能；第二，应用程序设计者可以使用 Stretch C 编译器开发自己的应用程序，而不需要专业的电子设计人员指导。

软件可配置处理器由传统的 32 位 RISC 处理器及一个可编程扩展指令集架构（Instruction Set Extension Fabric，ISEF）组成，并有一个 ALU 用于算术和逻辑运算，一个浮点部件（Floating-Point Unit，FPU）用于浮点操作。图 6.8 给出了

Stretch S6 软件可配置处理器。

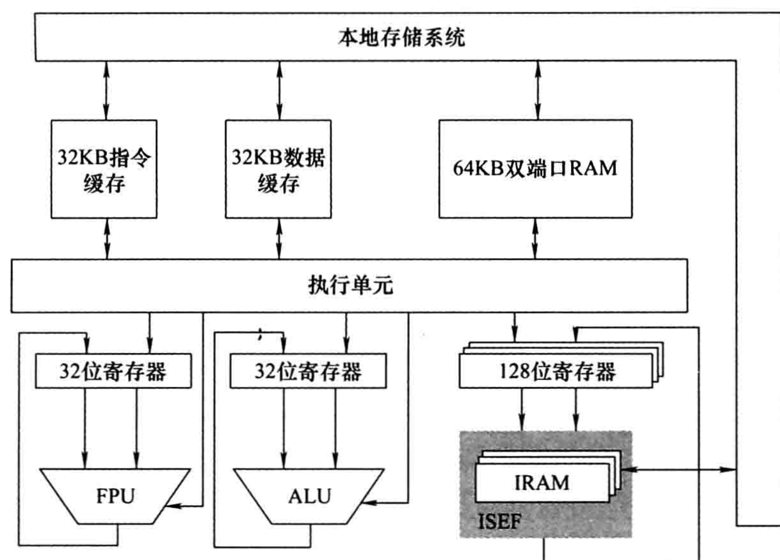


图 6.8　Stretch S6 软件可配置处理器[230]（IRAM 表示 ISEF 的嵌入式内存）

ISEF 由一组块组成，每个块包含一组 4 位的 ALU 和一组乘法单元，它们之间通过可编程路由结构连接。这种 4 位的 ALU 可以通过快速进位电路形成多达 64 位的 ALU。每个 4 位的 ALU 可以实现多达 4 个三输入的逻辑单元，并在寄存器中有 4 位用于扩展指令状态变量或者流水线。

该 ISEF 支持多个应用程序专用指令作为扩展指令。扩展指令集的参数由 32 个 128 位的寄存器提供。每个扩展指令可以读 3 个 128 位的操作数或写 2 个 128 位的结果。并有足够多的存取指令可以在寄存器、缓存及存储子系统之间传递数据，数据的位宽是 128 位。ISEF 通过扩展指令集可支持深度流水线。

除了上述存取模型，还有一组扩展指令可以在 ISEF 内部定义任意的状态变量，并存在寄存器中。组内的任何扩展指令都可以读取和修改状态变量，从而减少了大量的寄存器之间的信息交换。

除了这种软件可配置处理器，还有一种可编程加速器。它由一系列功能模块组成，这些功能模块通过专用的硬件来实现。这些功能块包括视频编码的运动估计、用于 H.264 视频的熵编码、基于高级加密标准（Advanced Encryption Standard，AES）和三重数据加密标准（Triple Data Encryption Standard，3DES）的加密操作，以及各种音频编码（如 MP3 和 AC3 格式等）。

为了开发一个应用程序，程序员要识别出需要加速的关键部分，并使用一个叫做 Stretch C 的变体 C 语言写一个或多个扩展指令，并作为函数在应用程序中访问。更多有关软件可配置处理器的应用程序匹配问题将在 6.6 节介绍。并且，

有关 Stretch S5 软件可配置处理器工具的研究将在本书 7.6.2 节和 7.7.2 节介绍。

6.6 可重构设备上的映射设计

可重构器件中的资源需要进行适当的配置才能实现一个特定的应用程序。下面看一下用于 FPGA 和软件可配置处理器上的映射设计方法。

典型的 FPGA 工具流程图如图 6.9 所示[56]。在传统的工具流程图中，如 VHDL 和 Verilog 这样的 HDL 被广泛用于工业设备中，用来描述在 FPGA 中实现

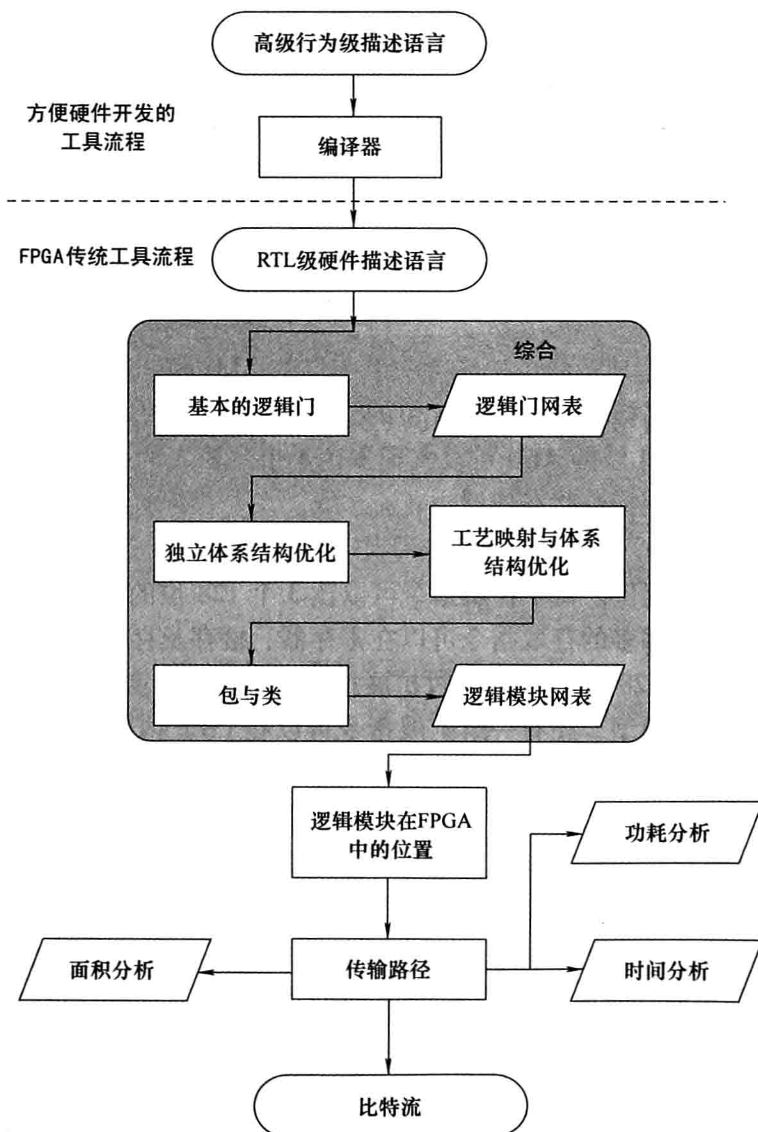


图 6.9 典型的 FPGA 工具流程图

的电路。这种电路的描述在 RTL 级，在每个时钟周期中设计具体的操作。然后这些描述在 FPGA 布线之前综合成网表。

在综合的第一阶段，识别 RTL 级设计的数据操作，如控制逻辑、存储模块、寄存器、加法器和乘法器等；然后加工成基本的布尔逻辑，如与、或、异或等。

接下来，对基本逻辑门组成的网表进行优化，这种优化是独立于 FPGA 的。这种优化通常包括布尔表达式最小化、消除冗余逻辑、缓冲区共享、重定时和有限状态机编码。再将优化的逻辑门网表映射到具体的 FPGA 架构中，如 Xilinx Virtex 设备，或者是 Altera Stratix 设备。还有基于具体架构的更深层的优化，如加法器进位链、移位寄存器的专用移位功能。综合的最后阶段把多个 LUT 和寄存器进行封装并集成为图 6.4 所示的模块。这种封装与集成使不同逻辑模块之间连接的数量达到最少。综合之后，映射到网表中的逻辑模块根据线路速度、可布线性及线长等不同的优化目标移植到 FPGA 上。逻辑模块的位置一旦确定，不同的 I/O、逻辑模块及其他嵌入单元之间的连接就可以映射在 FPGA 的可编程布线资源上。布线过程决定了应该用哪些可编程开关连接逻辑模块的 I/O 端口。最后，所有I/O、逻辑块和具体 FPGA 电路中的连线资源的可配置的比特流就形成了。

上面的描述涵盖了对细粒度 FPGA 资源的映射设计。许多可重构器件含有用于运算和存储的粗粒度的资源（见 6.5.1 节），因此 FPGA 设计工具需要将这些资源考虑进去。

为了提高生产率，FPGA 工具流中包括一些高级程序设计语言（图 6.9 上半部分所示），如 AutoPilot[272]、Harmonic[159]及 ROCC[248]。这些语言和工具使得应用程序开发人员即使不具备技术实现方面的知识也能进行设计。有些编译器能够从源代码中提取计算的并行性，来实现流水线的优化。这些工具通常会提高应用程序开发人员的生产效率，但会牺牲一定的产品质量。然而，由于器件容量和功能持续快速增加，应用程序开发人员的生产率可能会被优先考虑。

除了配置电路，有些工具还可以分析电路的延迟、面积及功耗。这些工具通常被用于检查电路是否满足应用程序开发人员的要求。

对于美国 Stretch 公司的软件可配置处理器，编译器既要确定执行单元，又要确定 ISEF，如图 6.8 所示。Stretch C 编译器[25]的第一阶段需要选择一种扩展指令集并使用各种优化，如常数传递、循环展开及字长优化等。除此之外，还要将许多运算符加入到平衡树中合适的位置；把乘法运算用多次移位和加法实现；共享不同指令所使用的资源。然后编译器产生两个主要的输出——指令头和延迟信息，包括 Xtensa 编译器的寄存器使用情况，以及从源代码中提取的用于在 ISEF 中[25]进行资源映射的结构网表。

Xtensa 编译器可以编译与扩展指令相关的应用程序代码。它使用 Stretch C 编译器产生的程序头和计时数据实现寄存器的分配及指令流调度的优化。然后将

这些结果与 ISEF 比特流关联起来。

在 Stretch 编译器中产生 ISEF 比特流的过程，与前面介绍的 FPGA 工具流程很类似。主要有以下四个阶段[25]：

- 映射（Map）阶段。为 Stretch C 编译器初始阶段提供的运算符产生执行模块。
- 置位（Place）阶段。为 Map 阶段生成的模块分配地址。
- 路由（Route）阶段。在网表上执行具体的由时间驱动的路由操作。
- 重新定时（Retime）阶段。移动寄存器来平衡流水线每个阶段的延迟。

连接器将应用程序的组件打包成一个可执行文件，该文件包含一个 ISEF 配置文档，可用于操作系统或运行时系统在动态重构过程中确定指令的位置。

6.7 特定实例设计

特定实例设计通常用于定制硬件和软件。其目标是对一个特定运算的实现方式进行优化，主要作用是提高速度、减少资源的使用、降低功耗，但其灵活性较差。

下面介绍三种自动化特定实例设计。第一种是常数合并，通过计算、传递静态输入值，从而消除不必要的硬件或软件。图 6.10 所示的特定实例的硬件设计被实例化为图 6.11 所示的具有特定过滤系数的设计。只有一个输入的常系数乘法器比两个输入的乘法器更小、更快，因此其使用效率更高。

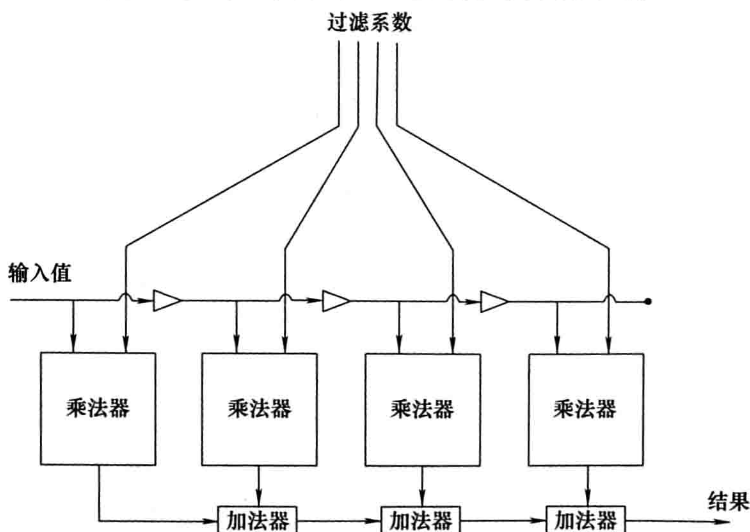


图 6.10 一个 FIR 滤波器包含两输入的乘法器[197]
（支持可变滤波器系数；三角形表示寄存器）

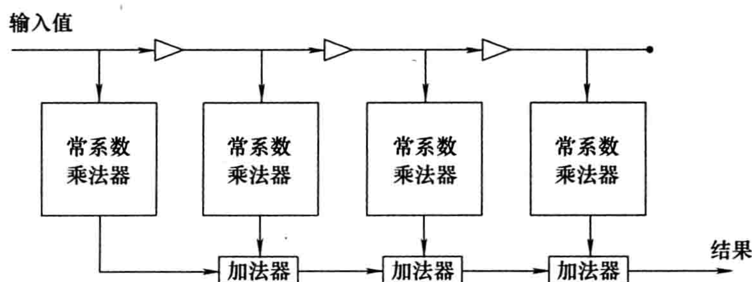


图 6.11　一个 FIR 包含单输入的常系数乘法器[197]（支持具体实例的滤波器系数）

可重构逻辑能实现一些专用设计，并将不同的专用设计应用于同一个设备上从而提高灵活性。在一些应用程序的实现中可重构逻辑比 ASIC 更高效。对其他应用程序，在一个具体问题实例的优化设计中，性能的提高可以使 FPGA 的性价比高于 ASIC。

特定实例设计的重要优势在很多应用程序中有所体现。例如，在有限脉冲响应（Finite Impulse Response，FIR）过滤器中，修改公共子表达式消除算法可以减少用于实现常系数乘法运算[180]的加法器的数量。与基于分布式算法的设计相比，特定实例设计可以使 FPGA 片的数量减少 50%，用于完全并行实现的 LUT 数量可以减少 75%。而且，这种 FIR 滤波器的整体动态功耗会降低 50%。

在运行时改变特定应用程序的设计不如改变电路的输入迅速，因为一个新的完整的或部分的配置加载进来，要花几十甚至是几百微妙的时间。因此，选择一个设计的实例化方法很重要。运行时配置的相关讨论将在 6.9 节展开。

第二种技术是功能修改。对一个应用程序实例，使用硬件或软件技术修改其功能，可以获得性能与资源使用率之间的平衡，以及满意的程序运行结果。

字长优化就是一个功能修改的例子。考虑到细粒度 FPGA 的灵活性，可以自动化实现一个进程，来寻找一个好的定制数据的表示方法。选择合适的字长和扩展 DSP 系统中的每个信号，对自动化的实施都很重要。并不像基于微处理器的实现那样使用处理器的硬连线结构要优先考虑字长，基于 FPGA 的可重构计算允许自定义每个变量的大小，因此可以在数字准确性、设计尺寸、速度和功耗方面达到最优。

第三种技术是对结构进行修改，包括通过修改硬件结构和软件结构来优化具体的应用程序实例，如支持相关的定制指令。下面将详细地介绍这一方法。

表 6.3 给出了特定实例设计说明，是针对以上三种技术的。特定指令设计的更多信息可在其他地方找到[197]。

表 6.3 特定实例设计说明

技 术	目 标	实 例	实例中的作用
常数合并	对静态输入的值进行优化	FIR 过滤器[180]	动态功耗减少 50%
功能修改	优化函数运行结果	字长优化[62]	减少功耗 87%
结构修改	优化应用程序实例的体系结构	定制指令处理器[77]	速度提高 72%，面积增加 3%

6.8 可定制软件处理器的一个实例

本节介绍一个含有定制指令集的被称为 CUSTARD 的多线程软件处理器[77]。它涉及了前面的一些实质性内容。即，一个指令处理器怎样定制，并通过修改指令结构以支持不同的线程与定制指令；确定重构技术的相关工具流。

CUSTARD 在同一个处理器中可以支持多个上下文。一个上下文是指线程执行的一个状态，尤其是寄存器、堆栈及程序计数器的状态。在硬件层面上支持线程有两个好处：第一，上下文切换（即改变活动线程）可以在一个周期内完成，使得单处理器交错执行独立的线程时开销很小或没有开销；第二，上下文切换可用于隐藏延迟，否则一个单线程可能会忙等。

支持多线程的缺点主要是每个上下文都需要一些额外的寄存器。幸运的是，现在的 FPGA 有很多片上存储模块可用于实现较大的寄存器堆。处理器的控制部分需要添加额外的复杂逻辑，并且当前线程的状态必须在流水线的每一级中记录下来。然而很多流水线和功能单元在多个线程之间是共享的，因此在多处理器结构中，可以考虑降低这些流水线和功能单元的面积。

CUSTARD 处理器实例是使用参数化的模型生成的。图 6.12 给出了CUSTARD 微体系结构，包括了参数模型的一些关键组成部分。这种模型在实例化一个可综合的硬件描述及配置精确周期模拟器时都可以使用。

CUSTARD 基础结构是一个典型的软件处理器，支持完全旁路与互锁的四级流水线。这是一个 load/store 的 RISC 体系结构，支持 MIPS 定点指令集。它可以增加流水线，同时可以将 MIPS 指令操作码的剩余空间作为定制指令的操作码。

CUSTARD 的定制需要四组参数。第一组包括多线程支持，可以指定线程的数量及线程的种类，块多线程（Block Multithreading，BMT）或交替多线程（Interleaved Multithreading，IMT）。第二组包括定制指令集和流水线执行阶段相关的数据通路及定制存储块。第三组包括旁路与互锁结构。无论是在分支延迟槽，还是在加载延迟槽过程中，旁路路径都是必要的。第四组包括寄存器堆，给出了寄存器的数量与寄存器堆的端口数。

CUSTARD 结构支持 BMT 和 IMT 两种多线程结构。这两种多线程结构同时用

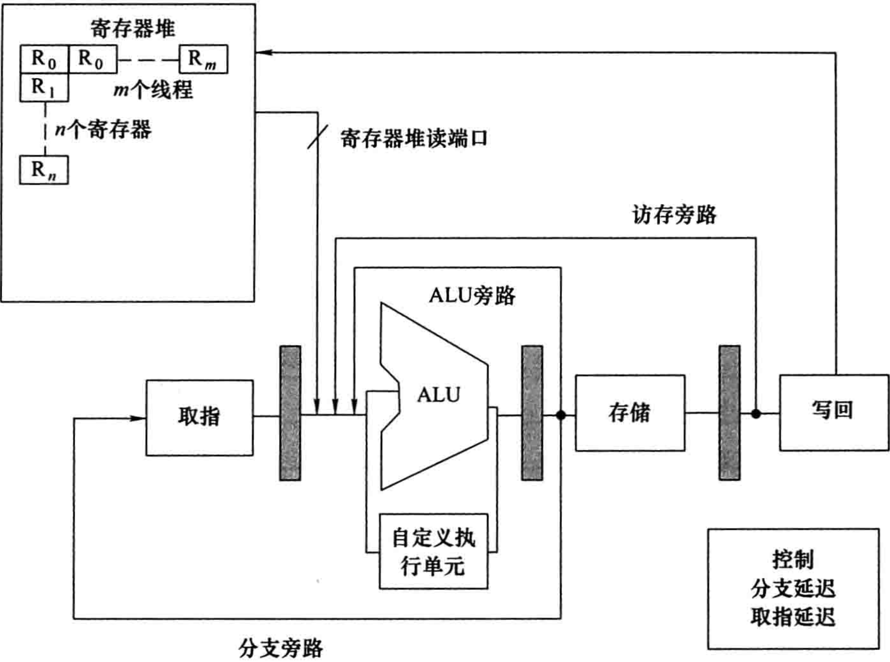


图 6.12　CUSTARD 微体系结构（具有多线程、寄存器堆和参数的旁路路径）

于维持多个独立线程的上下文（寄存器、程序计数器的状态等）。线程的类型因上下文切换引起的环境的不同而有所差别。

BMT 触发上下文切换，是由当前活跃进程在运行时产生的一些事件引起的，如缓存缺失、显式的吞吐量控制或者是一些延迟较长的定制指令开始运行等。当只有一个线程可运行时，BMT 处理器就跟传统的单处理器一样了。当多个线程可运行时，可以通过上下文切换隐藏所有活跃线程中的延迟。上下文切换在流水线的执行阶段触发，因此最后取得的一条指令在新的活跃线程中必然会被刷新和重填。

IMT 在每个周期都执行强制的上下文切换，从而导致多个线程的交替执行。IMT 允许流水线的简化，因为如果有足够多的线程，流水线的一些阶段可以保证存在独立的指令。因此，IMT 可以消除或者通过旁路和互锁网络的简化设计减少流水线冲突。CUSTARD 处理器可以利用这一特点，通过有选择地消除旁路路径，最终实现对具有特定线程的处理器的优化。

表 6.4 给出了前递路径（见本书图 5.12）、单线程中可优化的互锁关系、BMT 参数及 IMT 参数的总结，包括了每个多线程结构中旁路与互锁结构的定制方法。BRANCH、ALU 及 MEM 的旁路路径如图 6.12 所示。IMT 列显示了旁路和互锁网络中的元素可以怎样根据可运行线程的数量消除掉。例如，当只有两个线程可运行时，ALU 前递逻辑可以消除。当两个线程可以同时运行时，任何进入流水线 ALU 阶段的指令都独立于其他 ALU 阶段的指令。“ * ”标记的情况指约

束指令的输入顺序，需要删除互锁关系；相关参数可在编译器中使用，然后用于指令的调度。

表 6.4 前递路径（见本书图 5.12）、单线程中可优化的互锁关系、BMT 参数及 IMT 参数的总结

关闭线程的数量	BMT≥1	IMT 2	IMT≥4
分支前递		√	√
ALU 前递	√	√	
MEM 前递		√	
分支延迟	√*	√	√
取数延迟	√*	√	√

* 对结构中可以改变编译器调度的元素进行优化从而减少冲突。

多重上下文是由多重寄存器堆支持的，这些寄存器堆在 FPGA 上以双端口 RAM 的形式实现。每个寄存器堆通过寄存器号和线程的 ID 共同索引。每个寄存器堆是可参数化的。这种参数化可以根据端口数和每个线程中寄存器的数量决定。寄存器堆中端口数的增加使得含有大量操作数的定制指令可以通过编译器选择。

将并行命令式语言编译到硬件中[191]的方法可用于实现可参数化的处理器。CUSTARD 为处理器的参数化提供了一个架构，以及一个实现成硬件的方法。对一个给定的应用程序，相关的编译器会产生 MIPS 定点指令与定制指令，从而实现 CUSTARD 的优化。表 6.5 给出了一些基准程序定制指令。

表 6.5 一些基准程序定制指令

基准程序	定制指令（输入寄存器为 $r_0 \sim r_3$，立即数值为 imm_0）	使用数量	延迟/周期	BRAM/B
Blowfish	$LUT(r_0 >> 24) + LUT(r_1 >> 16)$ $LUT([r_0 >> 8] \& 255)$	2 2	1 1	1024
Color space	$([r_0 >> 8] \& 0xFF) \mid (r_1 \& 0xFF00)$ $\mid ([r_2 << 8] \& 0xFF0000)$	1	1	32
DCT	$LUT(r_1) + r_2 * (r_0 << 8)$ $LUT(r_1) + r_2 * ([r_0 \& 255] - 128)$	65 65	2 2	64
Edge detect	$LUT(r_0 + 1 + imm_0)$	3	1	64
Susan	$LUT(r_0)$	31	1	516
AES	$LUT(r_0) \wedge LUT(r_1 >> 8) \wedge LUT$ $(r_2 >> 16) \wedge LUT(r_3 >> 24)$	64	1	1024

注：输入寄存器 $r_0 \sim r_3$ 表示分配的通用寄存器。LUT（a）为在查找表中以 a 为地址在专用的 BRAM 中查找。使用数量表示通过指令在基准汇编代码中使用的次数显示重用的程度。延迟表示运行结果可用于旁路或寄存器堆前，所需要执行的周期数。

图 6.13 给出了一个具体应用程序的 CUSTARD 定制处理器的工具流程图。定制指令是基于类亚指令（similar sub-instruction）[77] 技术产生的。发现定制指令之前，在预优化阶段会执行标准的源代码级优化，如进行循环展开，从而发现循环的并行性。选择定制指令之后，重新调度定制指令与基本指令以使流水线停顿达到最小。这个调度阶段是可以参数化的，可以支持 CUSTARD 多线程模式引起的微结构变化。

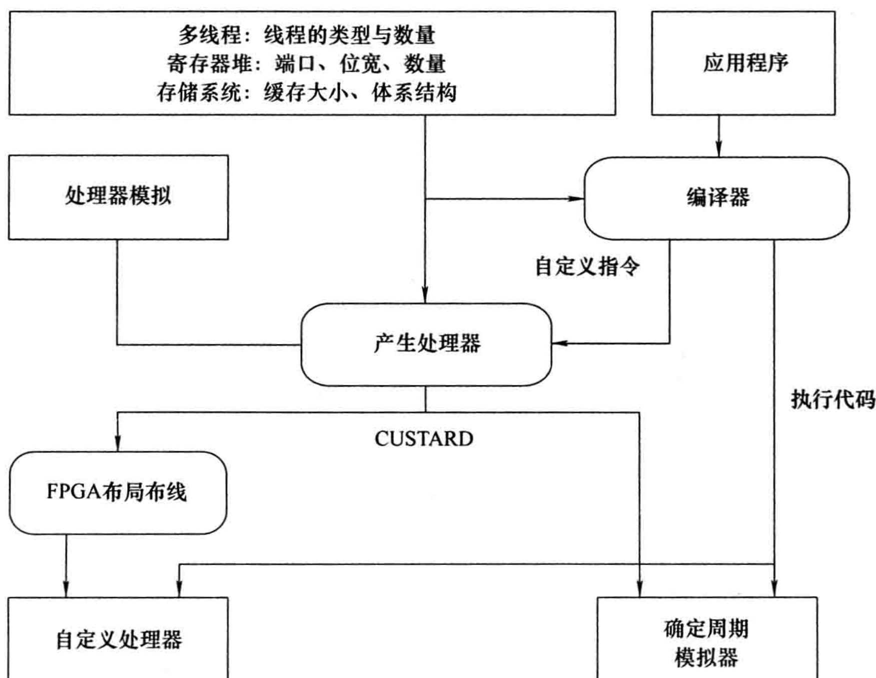


图 6.13　一个具体应用程序的 CUSTARD 定制处理器的工具流程图

编译结果包括硬件数据通路，以实现定制的指令，还包括在定制处理器上定制运行的软件。定制指令的数据路径被添加到 CUSTARD 处理器上，然后修改解码逻辑，将新指令映射到不用的指令操作码部分的地址空间上。

这里有一个基于 SimpleScalar 架构[30] 的精确周期模拟器。这个模拟器可在处理器硬件描述中直接修改，并能够模拟一个参数化的存储系统。

在 Xilinx XC2V2000 的 FPGA 上实现的 CUSTARD 处理器开发了五个基准应用程序：Blowfish、Colourspace、AES、DCT（Discrete Cosine Transform，离散余弦变换）和 SUSAN。结果显示，与单线程处理器相比，IMT4（四线程的 IMT）和 BMT4（四线程的 BMT）面积分别增加了 28% 和 40%，却可以允许四个线程交替执行，并且不需要软件开销。而且与没有定制指令的相同结构相比，定制指令使性能平均提高 72%，面积平均只增加 3%。CUSTARD 使 AEX 的速度加快了 355%。

没有定制指令的 IMT 处理器最大时钟频率比 BMT 处理器和单线程处理器分别高了41%和5%。时钟周期的数量平均减少了10%。IMT 处理器通过交错执行独立线程隐藏了流水线延迟。估计这种较低的（低10%）改善是由定制指令产生的延迟较短引起的（见表6.5），在每种情况下的延迟最多只有两个时钟周期。由于寄存器堆的端口数量有限，建立延迟较长的指令是不可能，所以希望深度流水线处理器或者定制的浮点指令可以创建足够长的延迟以利于这部分性能的提高。然而，IMT 处理器可以通过消除 ALU 周围的旁路逻辑获得较高的最大时钟频率。通过时序分析器显示，这些 ALU 旁路逻辑在 BMT 和单线程处理中是很有必要的。

6.9 重构

重构的动机有很多：第一，分享当前没有需求的资源；第二，需要进行升级来支持新功能、新标准或者新协议；第三，在运行时调整硬件。

运行时的重构为实现很多应用程序提供了可能，如自动化系统[35]、高性能计算[81]、网络互联[161]、视频处理[211]和自适应 Viterbi 解码[241]。下面介绍的方法是在具体的技术中提取出来的，重点讲述运行时重构设计的开销分析。

6.9.1 重构的开销分析

修改一个体系结构以适应具体的应用程序，其最终目标是通过使用具体的应用程序优化技术提高性能。然而，通过修改体系结构获得的性能的提高必须超过重构所付出的代价。考虑一个软件函数 f ( )，在一个具有标准通用指令集的体系结构上运行要花时间 t_{st}。修改后，支持定制指令，运行时所需要的时间是 t_{ci}，可表示为原来执行时间的一部分，即 αt_{si}，其中 $0<\alpha<1$。对函数 f ( ) 运行所使用的设备进行重构需要时间 t_r。定义一个可重构比例系数 R 通过分析相关的开销判断对新的体系结构进行重构是否是值得的，有

$$R=\frac{t_{si}}{t_{ci}+t_r}=\frac{t_{si}}{\alpha t_{si}+t_r} \tag{6.1}$$

利用可重构比例系数 R 可以衡量重构新体系结构所带来的好处。相对当前的实现方式来说，R 提供了一个重构后新的实现方式的改善因子。$R=1$ 的点是临界点：如果 $R>1$，重构将是有益的。但必须注意，$R=2$ 并不代表系统整体性能提高两倍。最大重构比例系数 R_{max} 是对绝对性能提高的程度的评估标准，计算时不考虑重构时间。对一个定制指令来说，R_{max} 是 R 的最大可能值。它由下面的公式确定：

$$R_{\max} = \lim_{t_r \to 0} R = \frac{t_{si}}{t_{ci}} = \frac{1}{\alpha} \tag{6.2}$$

可重构比例系数 R_{pot} 的最大可能值是对性能可以提高的最大比例的测量标准，它显示了一个定制指令达到可重构阈值的快速程度，有

$$R_{pot} = \lim_{\alpha \to 0} R = \frac{t_{si}}{t_r} \tag{6.3}$$

R_{pot} 为重构的粒度和函数的大小提供了一个参考。一个函数的 R_{pot} 值越大，越容易修改，有

$$\frac{t_{ci}}{t_{si}} = \frac{R_{pot} - 1}{R_{pot}} \tag{6.4}$$

因此，$R_{pot}=5$ 意味着原来指令执行速度是现在的 4/5，重构可以执行；$R_{pot}=4$ 意味着原来指令执行速度是现在的 3/4；同理，$R_{pot}=3$，意味着速度是现在的 2/3，以此类推。

如果函数 f ( ) 的 $R_{pot}=1$，那么函数的修改及随后的重构是无意义的。

可重构比例系数也可以用 FIP 的时钟数目描述。考虑函数 f ( )，执行需要花 t_{si} 个时间单位，执行该函数要用 n_{si} 个时钟周期，每个时钟周期的时间是 T_{si}。应用程序执行一次，该函数被调用 F 次。t_{ci} 也是这样，则有

$$\begin{aligned} t_{si} &= n_{si} T_{si} F \\ t_{ci} &= n_{ci} T_{ci} F \end{aligned} \tag{6.5}$$

重构时间 t_r 可以用重构时钟周期 T_r 与周期数 n_r 的乘积来表示。重构时钟周期 T_r 与平台相关，而与在可编程器件上进行设计的周期时间无关。重构所需要的时钟周期数目由完全重构还是部分重构决定。完全重构时，n_r 是一个与具体可编程器件相关的常数；部分重构时，n_r 会因重构时修改的次数不同而有所差异。这里有一个因子 τ 代表某些重构所引起的开销，如重构前停止设备、重构后打开设备的开销。重构时间为

$$t_r = n_r T_r + \tau \tag{6.6}$$

对现在的可编程设备来说，打开和关闭时间可以忽略不计。例如，在 Xilinx Virtex 设备中，这个值可以小到重构一个最小的原子可重构单元所用时间的 10%。

其他的开销时间包括保存与恢复处理器的状态所用的时间。在最极端的情况下，处理器的状态包括处理器中的所有存储部件，如寄存器堆、流水线中的寄存器、程序计数器或者是缓存。通过恢复处理器的状态，处理器可以返回到状态被保存时的环境中。

将式（6.5）与式（6.6）带入式（6.1），可重构比例系数变为

$$R = \frac{t_{si}}{t_{ci} + t_r} = \frac{n_{si} T_{si} F}{n_{ci} T_{ci} F + n_r T_r + \tau} \tag{6.7}$$

该式可用于产生一个曲线。它显示随着 F 的增大，R 的变化趋势。其中，F 代表函数 f（）被调用过的次数。当 $R>1$ 时，重构是有益的。关于这种方法的更多信息可以在有关灵活自调整指令处理器（the adaptive flexible instruction processor）[213]的描述中发现。

6.9.2 平衡分析：重构的并行性

下面讲述一个可重构设备的简单分析模型[37]：重构面积越大，重构时间越长。对于那些在同一个任务中对数据单元做重复独立处理的进程，可以顺序执行或并发执行，增加这些程序的并行性可以减少处理时间，但会增加重构时间。下面的模型可以帮助找到一个折中的方案，即整个运行时间最短的并发度与重构时间之间的折中。

实现重构主要考虑三个属性：性能、面积、存储空间大小。一个存储单元数据的运行时间是 t_p，面积是 A，重构时间是 t_r，存储空间大小是 Ψ，运行需要 s 步，并行性是 p。

可以确定程序的参数：需要的数据吞吐量是 Φ_{app}，一系列的重构需要处理 n 个存储单元的数据；可重构器件的最大可使用面积为 A_{max}，接口的数据吞吐量是 Φ_{config}。

在易失性的 FPGA 上进行设计，需要额外的空间用来存储原始配置信息的比特流数据。使用局部运行时的重构进行设计时，还需要额外的空间存储可重构模块的预编译配置信息。考虑 A 表示 FPGA 片（如 CLB）上一个可重构模块的大小；Θ 表示具体设备的参数，即配置一个 FPGA 片所要的字节数；可重构模块的局部配置信息所需要的存储空间大小 Ψ 直接与其面积 A 有关：

$$\Psi = A\Theta + h \approx A\Theta \tag{6.8}$$

式中，h 为配置信息头部的大小，但大多数情况下，可以被忽略，因为该信息头部通常很小。

运行时重构的时间开销通常有多个部分组成，如调度时间、保护现场与恢复现场时间及重构进程本身的开销时间。考虑的这种情况没有调度开销，因为需要的模块直接被加载；也没有现场需要保存和恢复，因为一旦一组数据信号通过，信号处理模块就不需要保存状态信息。重构时间与局部配置信息的大小是呈比例的，其计算式如下：

$$t_r = \frac{\Psi}{\Phi_{config}} \approx \frac{A\Theta}{\Phi_{config}} \tag{6.9}$$

式中，Φ_{config}为配置数据的传输速率，以每秒传递的字节数计算。这个参数不仅取决于配置接口的数据传输速度，还与控制器及存储配置信息的存储部件的传输速度有关。

可以区分哪些运行时的重构现场的数据在重构时不需要缓存，哪些需要缓存。对于后面这种情况，可以通过重构时间 t_r 及数据吞吐量 Φ_{app} 计算缓存大小：

$$B = \Phi_{app} t_r = \frac{\Phi_{app}}{\Phi_{config}} \Psi \tag{6.10}$$

表 6.6 给出了不同功能和重构时间下的缓存大小。可以看到，数据在通过接收器时传输率减少了。因此，调用时重构在接收器链的末端更容易实现。缓存大小与通信带宽及重构的持续时间呈正比。

表 6.6　几种不同功能和重构时间下的缓存大小

功　能	数据吞吐量	给定配置时间的缓存大小		
		100ms	10ms	1ms
降频转换，16bit	800Mbit/s	80Mbit	8Mbit	800Kbit
降频转换，14bit	700Mbit/s	70Mbit	7Mbit	700Kbit
UMTS 解调	107.52Mbit/s	10.75Mbit	1.07Mbit	107Kbit
GSM 解调	7.58Mbit/s	758Kbit	75.8Kbit	7.58Kbit
UMTS 错误纠正	6Mbit/s	600Kbit	60Kbit	6Kbit
GSM 错误纠正	22.8Kbit/s	2.28Kbit	228bit	22.8bit
UMTS① 解密	2Mbit/s	200Kbit	20Kbit	2Kbit
GSM② 加密	13Kbit/s	1.3Kbit	130bit	13bit

① UMTS：通用移动通信系统，Universal Mobile Telecommunication System。

② GSM：全球移动通信系统，Global System for Mobile Communication。

缓存可以用片上或片外的资源实现。现在，大多数 FPGA 有快速的嵌入式 RAM 可用于实现先进先出缓存。例如，Xilinx 的 Virtex-5 FPGA 含有 1 ~ 10 Mbit 的 RAM 块。更大的缓存必须用片外存储资源实现。

运行时可重构系统的性能由重构时间决定。如果可重构的硬件设备用作软件中函数的加速器，尽管重构会有开销，但整体性能会有所提高。这里所考虑的情况是，通过重构来支持多种硬件功能从而提高灵活性、减小面积。但这种情况下，相对于没有重构的设计来说，会损失一定的性能。在处理器上重构所需要的时间通常比上下文切换所需要的时间长，因为有很多配置信息需要加载到设备中。与静态设计相比，重构设计的效率 I 可以这样描述：

$$I = \frac{t_{static}}{t_{reconf}} = \frac{n t_p}{n t_p + t_r} = \frac{n}{n + \frac{t_r}{t_p}} \tag{6.11}$$

通过处理更多的不同结构之间的数据，或者是提高重构时间与处理时间的比率，可重构系统的效率变得更高。下面提出一个考虑并行处理时间和重构时间的更详细的分析方法。许多应用程序可以在小而慢的顺序执行方式与大而快的并行

或流水线执行方式之间权衡。FIR 过滤器、AES 加密和坐标变化数字计算机（Coordinate Rotation Digital Computer，CORDIC）就使用了这种分析方法。

图 6.14 给出了一个需要四步（$s=4$）才能实现的算法的时空图。该算法与处理时间、面积及重构时间有关。每个已知数的处理时间 t_p 与并行度 p 呈反比。可以通过以下式计算：

$$t_p = \frac{t_{p,e}s}{p} \tag{6.12}$$

式中，$t_{p,e}$ 为一个元素的基本处理时间；s 为算法执行的步数；p 为程序的并行度。并行执行加快了数据的处理速度，但使重构变得缓慢。因为并行处理所需要的面积比串行处理的大，并且重构时间直接与面积呈正比，如式（6.9）。重构时间 t_r 直接与并行度 p 呈正比，这里 $t_{r,e}$ 表示一个处理单元所需要的基本重构时间，有

$$t_r = t_{r,e}p \tag{6.13}$$

现在可以计算一个规模为 n 个数据项的程序的运行时间

$$t_{\text{total}} = nt_p + t_r = \frac{t_{p,e}sn}{p} + t_{r,e}p \tag{6.14}$$

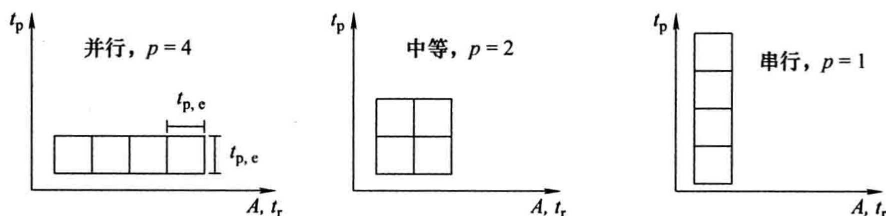


图 6.14　一个需要四步（$s=4$）才能实现的算法的时空图

图 6.15 给出了不同规模 n 及不同并行度 p 的标准处理时间，反映了并行性对运行时间的影响。考虑一个算法，其执行步数 $s=256$，通过观察可知，过滤器可以有 200 甚至更多个指令。每个数据的处理时间是标准的，假设每个处理部件的重构时间 $t_{r,e}$ 是基本运行时间 $t_{p,e}$ 的 5000 倍。在不同的应用程序及目标设备中，这个值是不同的，但对当前设备来说，至少在数量级上是正确的。可以看到，完全串行执行对一些规模较小的运算比较好。在这种情况下，较短的配置时间比较长的处理时间更重要。然而由于受配置时间的影响，整体的运行时间仍然很长。对于规模较大的运算进行完全并行设计比较好，因为运算时间主要是处理时间，重构时间所占比例很小。在中等规模的运算中，可以通过调整并行度来优化处理时间。

为了寻找最理想的并行度，计算式（6.14）的关于 p 的偏导数：

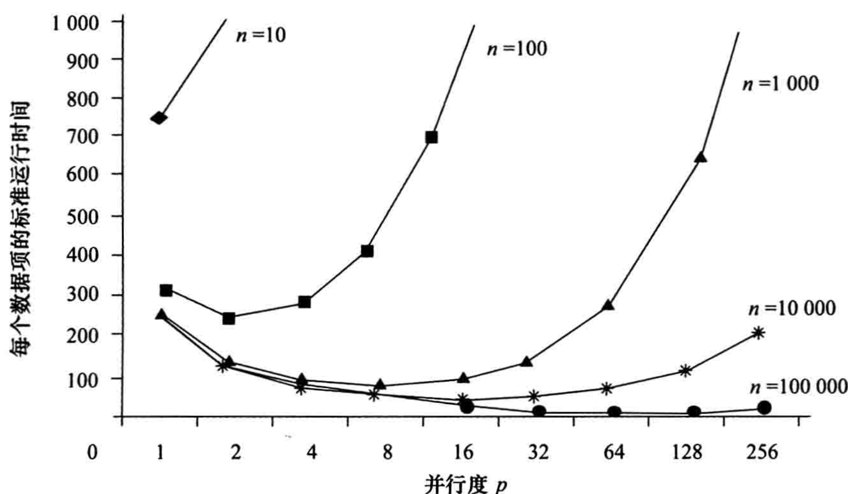


图 6.15　不同规模 n 及不同并行度 p 的标准处理时间（执行步数 $s=256$，假设 $t_{r,e}=5000t_{p,e}$）

$$\frac{\partial t_{total}}{\partial p}=\frac{t_{p,e}sn}{p^2}+t_{r,e} \tag{6.15}$$

为了寻找最小值，令式（6.15）为 0，解得

$$p_{opt}=\sqrt{\frac{snt_{p,e}}{t_{r,e}}} \tag{6.16}$$

结果 p_{opt} 通常是浮点数，不能用来表示并行度。为了确定 P 的实际值，可根据表 6.7所示指定 p_{opt}。

表 6.7　根据 p_{opt} 的值确定 p 的实际值

$0<p_{opt}\leqslant 1$	完全串行执行，$p=1$
$1<p_{opt}<s$	选择 $s/p\in Z$ 并且 $\|p_{opt}-p\|$ 最小
$s\leqslant p_{opt}$	完全并行执行，$p=s$

确定了使总体运行时间达到最优的并行度之后，就可以减少每个工作负载的总体处理时间，从而使性能达到最好。但仍然有必要检查这种实现方法是否满足了应用程序对吞吐量 Φ_{app} 的需求，有

$$\frac{n}{t(p)_{total}}=\Phi_{hw}\geqslant\Phi_{app} \tag{6.17}$$

最终需要的面积 A 必须小于总共可用的最大面积 A_{max}。总之，根据这种模型实现一个优化，必须按下面的步骤执行：

1. 从应用程序中得到 Φ_{app}、s 和 n。
2. 从使用的技术中获得 Φ_{config}。
3. 展开设计并确定 t_p 和 A。

4. 分别使用式（6.9）、式（6.12）、式（6.13）计算 t_r、$t_{p,e}$、$t_{r,e}$。

5. 用式（6.16）计算 p_{opt}并根据表 6.7 查找可行值。

6. 用式（6.14）计算 t_{total}，用式（6.17）检查吞吐量。

7. 以第 5 步中得到的并行度 p 实现设计，并检查实际的吞吐量是否满足要求。

8. 用式（6.10）计算缓存大小 B，并检查是否满足 $A \leqslant A_{max}$。

以上方法适用于很多应用程序和目标技术；使用这些方法能找到使性能达到最好的设计方案。为了找到满足吞吐量需求的最小的设计，可以尝试使用较小 p 值并用式（6.17）检查是否符合要求。

这种方法还可以扩展以处理可重构设计的能源效率问题[38]；相对于没有重构的设计来说，一个重构的 FIR 过滤器能源利用率会提高 49%，面积使用率提高 87%。

6.10 总结

定制技术可以以多种方式用于 ASIC 和重构技术中。这里对这些技术进行了综述，并展示了特定实例设计和定制指令处理器是怎样利用可定制性的。

随着技术的进步，以下两种影响变得越来越重要：

1. 集成电路封装成本迅速增加，使 ASIC 难以支撑。

2. SoC 设计与验证的复杂度持续增长。

本章所讨论的技术能够直接处理这些问题：不同程度的预加工定制（prefabrication customization）可以减少对 ASIC 技术的需求和设计复杂性。像 FPGA 这样的重构技术以后加工定制（postfabrication customization）的形式提供了相当程度的灵活性，但要牺牲一定的速度、面积及功耗。

定制与可配置性不仅广泛用于工业系统中，而且应用于学术研究领域。它们在 SoC 设计重用[215]、可综合数据通路架构[253]、微处理器动态扩展[48]、可定制多处理器[90]和其他方面都有新的进展。而且，动态可重构处理器已经开始在工业中使用，如日本松下公司的 D-Fabrix、日本日电电子（NEC Electronics）公司的 DRP-1、日本日立（Hitachi）公司的 FE-GA[17] 等。据说，ARM 处理器及其互联技术[74]，如 ARM 单元库和 AMBA 互联技术将会在 Xilinx FPGA 体系结构中使用与完善。毫无疑问，未来几年内，定制与可配置性将继续在电子系统中扮演重要角色。

6.11 习题

1. 从多种应用程序角度举例说明后加工定制的好处。

2. 一些可重构设备支持流水线互联。那么流水线互联的利弊是什么?

3. 一些 FPGA 厂商通过生产结构化的 ASIC 实现 FPGA 设计，实际上消除了可重构性。它们为什么要这样做?

4. 早期的 FPGA 只包含同种细粒度逻辑单元阵列，而现在有很多都是不同种类的。除了细粒度逻辑单元，现在的 FPGA 还包含一些可重构存储模块、乘法器甚至是处理器核。怎样解释 FPGA 体系结构的这种演化?

5. 一个机器 M 的部分指令集可通过协处理器 C 加速，并使速度提高 n 倍。

（a）假设一个应用程序 P 编译成在机器 M 上运行的指令，部分指令 k 属于协处理器执行的指令。那么在机器 M 上使用协处理器 C 其整体速度提高多少?

（b）（a）部分的协处理器 C 的成本是 M 的 j 倍。考虑可以用 C 对一个程序的指令加速，请计算要使 M 和 C 组成的系统应该比 M 快 j 倍，在协处理器 C 上运行的指令所占的比例最少是多少?

（c）M 的性能每个月提高 m 倍，那么要经过多少个月，才能在没有协处理器 C 的情况下，执行（a）部分所说的程序 P 的速度与 M 和 C 混合系统执行的速度一样快?

6. 怎样对式（6.7）进行一般化处理，从而可以覆盖 m 种定制指令。

7. 数据库搜索引擎使用运行时可重构的散列函数来减少要处理的资源数量。搜索引擎包括 p 个并行运行的处理器；每个处理器可以通过重构实现其中的一个散列函数。输入数据的总的单词数 w 被分成 e 个子集；每个子集用一个具体的散列函数处理，每个字中有一位用于标记匹配是否进行。标记位和相应的单词被存储在临时存储区，再用下一个散列函数在重构后的处理器中处理这些临时数据。每次迭代都更新标记位，处理过程一直运行直到数据被所有的 h 个散列函数处理完。令 T_h 表示散列函数处理器中的关键路径延迟，T_r 表示为支持不同的散列函数对处理器重构所需要的时间。访存需要 m 个时钟周期，每个单词平均由 c 个字母组成。考虑最坏的情况，即所有的散列函数都要执行，如果一个匹配没有发生就终止匹配过程，那么分析将会变得很困难。

（a）处理一个子集的数据需要多长时间?

（b）处理所有的数据需要多长时间?

（c）考虑每个字母占 b 位，那么需要多少个临时存储空间?

8. 为估算重构造成的开销对能源利用率的影响，可以建立一个分析模型，就像 6.9.2 节所述，相关应用程序包括以下内容：

- n，两次连续重构要处理的数据包或数据项的数量。
- s，算法中程序的执行步数。

一次重构的实现需要下面的参数：

- A，实现所需要的面积。

- p，实现中的并行度。
- t_p，一个数据包或数据项的处理时间。
- t_r，重构时间。
- P_p，处理时的功耗。
- P_c，计算功耗，是 P_p 的一部分。
- P_o，功耗开销，是 P_p 的一部分。
- P_r，重构时的功耗。

对可重构器件有以下两个参数：

- Φ_{config}，配置接口的数据吞吐量。
- Θ，每个资源或单位面积的尺寸。

耗电量是功耗和相关的持续时间造成的。

考虑计算引起的功耗 P_c 与并行度 p 成正比，存在一个不变功耗 P_o，重构引起的不变功耗 P_r：

(a) 处理 n 个数据项的计算功耗 E_c 是多少？

(b) 基于 P_o 的能源开销 E_o 是多少？

(c) 在重构时间与 p 呈正比时，重构的功耗 E_r 是多少？

(d) 每个数据项由于计算和重构所需要的总的功耗是多少？

(e) 寻找最优的并行度，以使每个数据项消耗的功耗最少。

第7章 应用研究

7.1 引言

本章描述了各种类型的应用，以此来阐明 SoC 设计中的机遇和利益的权衡。同时，也展示了如何应用前面章节所描述的一些技术。

首先，提出了一种设计 SoC 的方法。然后分析了高级加密算法（AES）中的几个简单的设计，以此阐述之前提出的技术。之后，将要探讨使用分析技术和原型技术的三维计算机图形学，以及这些技术在一个简化的 PS2 系统中的应用。最后，描述了静态图像、实时视频压缩方法及一些其他应用，以说明 SoC 的架构及其需求的多样化。

这里的讨论将**需求**和用以表现满足需求的**设计**进行了分离。**需求**包括需要什么；而**设计**包含了足够详细的实现细节，以便于评估设计是否满足需求。

7.2 SoC 设计方法

图 7.1 给出了一种 SoC 设计方法，它是前面各章原理的简化的 SoC 设计方法。本书第 2 章介绍了 SoC 设计中的五大问题：性能、芯片面积、功耗、可靠性和可配置性。这些问题提供了获取设计规格和运行时需求的基础。原始设计会逐步得到完善，并有望满足关键性的需求。通过处理跟内存（第 4 章）、互联（第 5 章）、处理器（第 3 章）、缓存（第 4 章）及定制化和可配置性（第 6 章）相关的问题，原始设计得以系统地优化。这个过程一直重复，直到设计满足性能和运行时的要求。本章会给出更多的详细说明。

然而，使用这种方法设计系统可能会是一个艰难的任务。系统设计通常要比组件或者处理器设计更具挑战性，而且经常需要经过多次迭代以确保：（1）满足设计需求；（2）接近最优，包括总成本（包括设计、生产制造及其他成本）和性能（包括市场规模及竞争力）。

通常，一个设计开始于一个最初的项目计划。如本书第 1 章所述，计划包括为产品开发做预算分配、时间表、市场估计（包含一份相关竞争产品的分析报告），以及目标产品性能和成本的估计。如图 7.2 所示，下一步是创建一个原始产品设计。此设计仅是一个很有可能满足目标产品性能和成本需求的雏形。再进

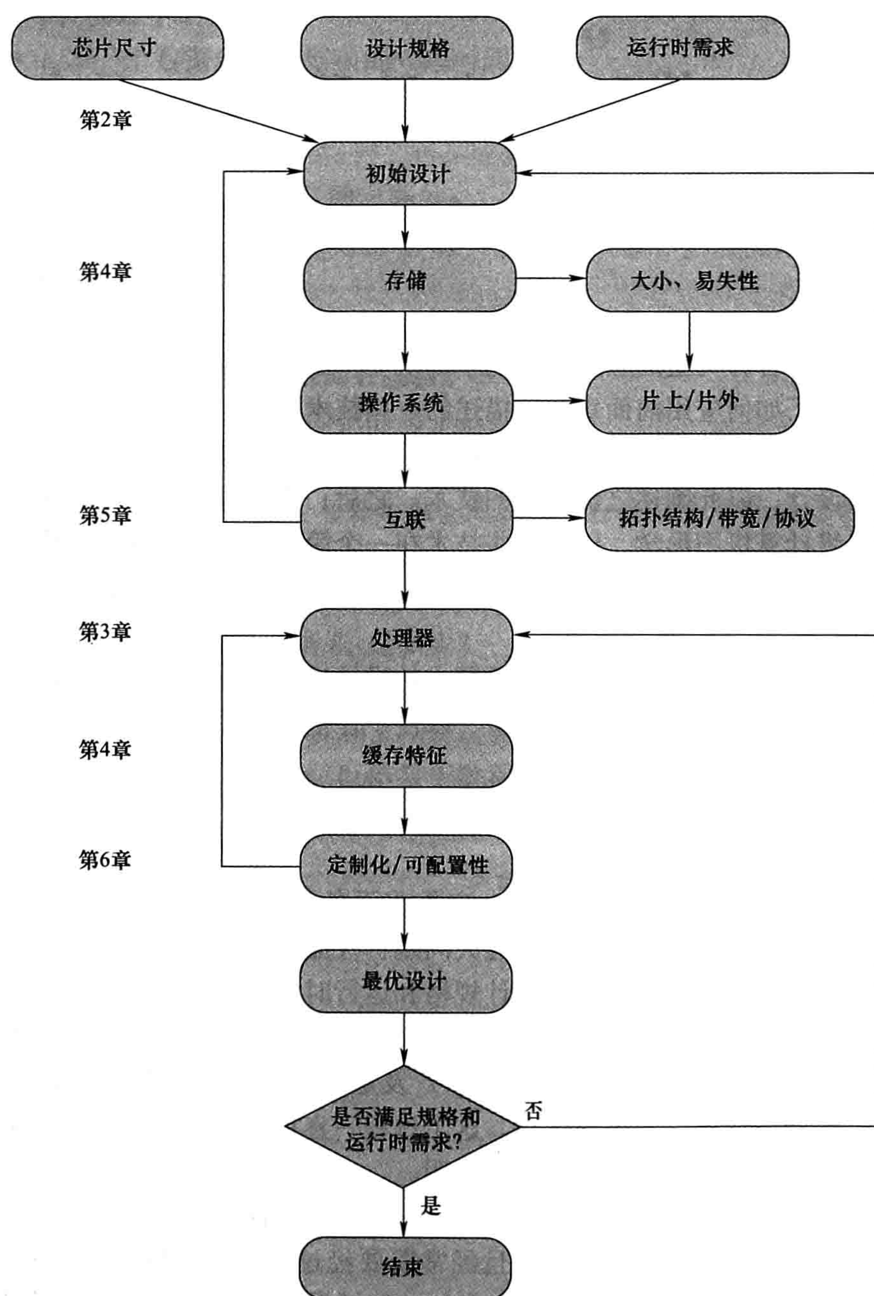


图 7.1 一种 SoC 设计方法

一步分析就能够知道该设计是否能满足需求。最初的分析中很重要的一部分是，完全理解性能和功能需求及它们之间的相互关系。应用的各方面均要进行详细说明和严格定义，并做出适当的分析和模拟模型。这些模型应该为应用提供一种平

衡性能和功能的方法，同时应该提供设计实现技术，这对满足运行时间需求是很重要的，如表7.1所示。

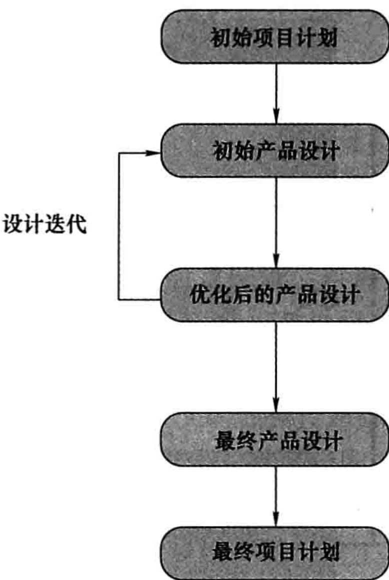


图7.2 系统设计流程

表7.1 某些视频和图形应用需要满足的运行时需求的限制

应用	实时性限制/(帧/秒)	其他限制
视频会议	16	帧大小，误差率，丢帧率
三维图形	30	图像大小，阴影，纹理，颜色

需求和设计

需求分析的输入通常是一份来自客户或市场调研的需求规格，输出则经常是设计的功能需求规格。对于大企业，需求说明书可能是为了评审而详细书写的。然而，对于小企业或者刚起步的企业，可能是以简短的电子表格形式撰写的。不论详细或简单，设计规格都是为了做好设计的评审、归档和验收工作，而且其内容必须清晰连贯。另外，在各个系统设计阶段，精确、可执行或者图解方式的说明，可以用来获取SoC的主要操作或者所选功能的数据流和控制流。这些描述能够帮助设计者理解功能需求。一旦理解了功能需求，这些描述就可以被映射到SoC中主要部件及其相互间交互的合理的实现中。

产品设计规格做好之后，就要着手提出一个初步的系统设计（见图7.3）。设计规格通常规定了产品的大致布局和编址问题。例如，是将所有元件放到一个芯片上还是将整个系统置于板卡上，以及操作系统、访存的总体大小、后备存储

器的选择。为了满足电路需求，将从以下方面继续完善最初的设计：

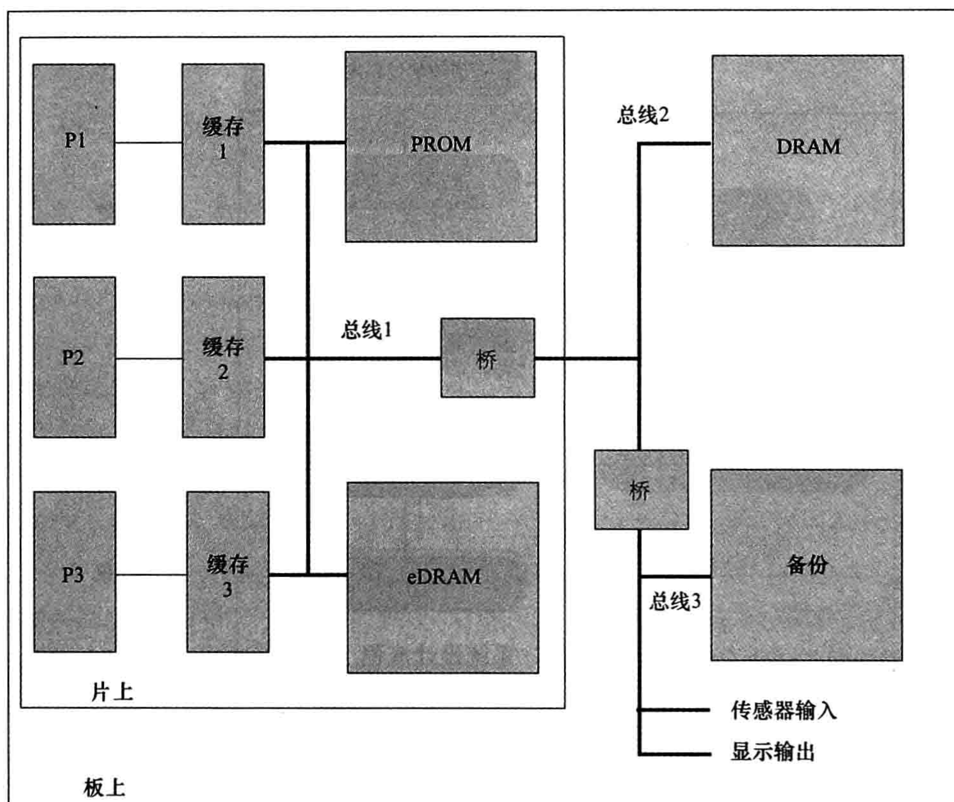


图 7.3　三个处理器 P1、P2 和 P3 的初始设计例子

1. 选择分配内存和后备存储器。这通常按照本书第 4 章的描述进行。

2. 一旦内存分配好，处理器就被选择好了。通常选择用一个简单基础的处理器来运行操作系统及管理应用控制功能。依据严格计算的特性，时间要求严格的进程可以被分配给专用处理器（如本书第 1 章和第 3 章讨论的 VLIW 和 SIMD 处理器）。

3. 内存和处理器的布局大体地定义了本书第 5 章所涉及的互联架构。现在需要确定的是带宽需求。设计规格和处理器目标性能主要决定了整体需求，而缓存可能会在满足规范方面充当一个重要的缓冲元件。通常，最初的设计假定互联带宽是足够与内存带宽相匹配的。

4. 分析内存因素，评估它们对延迟和带宽的影响。缓存和数据缓冲器的大小按照满足内存和互联带宽需求来选择。一些细节将会在后面提到，如确定总线延迟时，通常不需要考虑总线竞争。到此为止，处理器性能模型设计就完成了。

5. 某些应用需要进行外设的选择和设计，它们的设计也必须满足带宽需求。在7.5.1节和8.7节分别给出了几个外设的例子。7.5.1节的外设例子涉及数码相机的JPEG系统。8.7节的外设例子是关于未来自主SoC系统所用到的无线频率和光通信。

6. 确定整体成本和性能的粗略估计。

初始设计完成之后，开始进入设计的优化和验证阶段。这个阶段需要多种工具的支持，包括性能分析工具和优化编译器。所有部件及分配都要遵照降低代价(面积)、提高性能和功能的原则重新评估。例如，定制化和可配置性技术，如第6章所提到的自定义指令或者运行时重配置，可以被应用以提高灵活性或性能。再例如软件优化，如那些改善参考位置的软件优化技术也经常以较小的硬件成本换来较大的性能提高。应用的优化会影响到硬件和软件的重新分配，这将可能影响嵌入式软件程序[151]的复杂性及实时性操作系统[34]的选择。

完成每项优化之后，都需要分析该优化对准确度、性能、资源利用、功耗等产生的影响，并且需要对设计进行验证以确保正确性不会受到此次优化的影响[53]。这些分析和验证通常是通过系统级电子设计工具（见下框）和原型搭建的支持来实现。

系统级电子（Electronic System Level，ESL）设计和验证

关于ESL都涵盖哪些范围，似乎还没有一个标准的描述。维基百科对ESL设计和验证的定义是“一种新兴的首要专注更高抽象级别的电子设计方法学”。另一种ESL的定义是，在比较合理的性价比的基础上，利用适当的抽象化增加对系统的理解，提高成功实现功能的可能性[32]。目前开发的各种ESL工具都能够很好地支持在算法描述级跨硬件、软件边界地开发系统[18]。

当系统设计经过几次迭代趋于最优之后，另一个完整的项目计划就出现了，它可以帮助理解任何一个对固定成本规划所做的修改，以及处理系统集成和测试相关的问题。最后，基于最终的设计对产品市场进行评估，也可以对整体项目的收益做出评估。

7.3 应用研究：AES

下面以AES为例，来阐明如何应用前面所讨论的技术来探索满足特定需求的设计。

7.3.1 AES：算法及需求

AES[69]有三种规格：128位（AES-128）、192位（AES-192）和256位

（AES-256）。从原始数据到加密后的数据，整个过程包括一次初始轮，$r-1$ 次标准轮，以及一次最终轮。主要的转换过程包括以下几步（见图 7.4）：

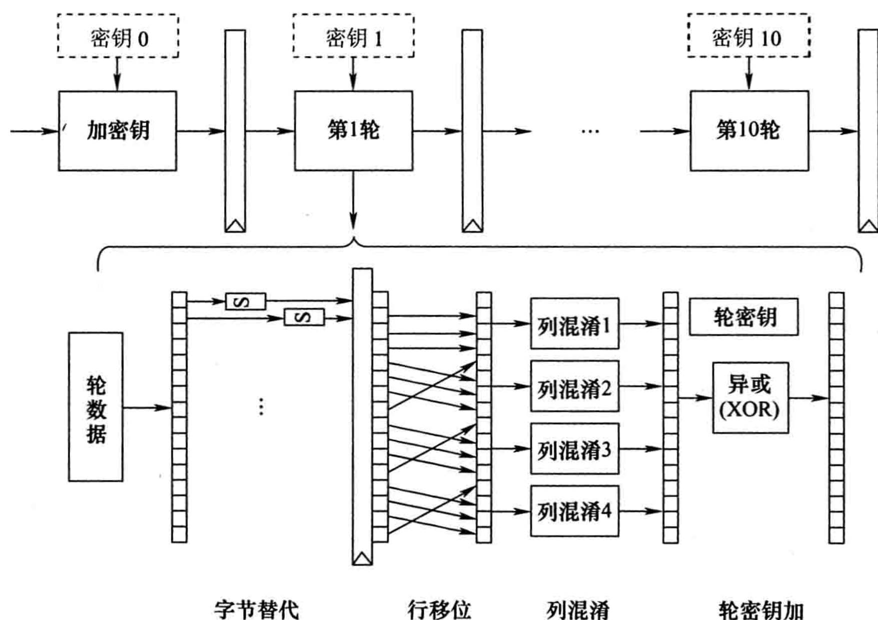


图 7.4 全流水 AES 架构[107]

- 字节替代　使用专用的设计替换盒子（S-BOX）将输入块以字节为单位进行转换。
- 行移位　输入的字节被重新排列成四行。然后根据行号按照预定的步骤对每一行进行旋转。
- 列混淆　通过基于有限域 GF（2^8）上的多项式乘法对重新排列的四行结构的每一列进行变换。
- 轮密钥加　输入与每次循环的密钥相异或。

初始轮包括一次**轮密钥加变换**；标准轮包括以上四步变换；最终轮包括以上四步中**列混淆变换**之外的其他三步变换。另外，解密过程就是以上运算的逆变换。轮变换可以并行执行以加快加密过程。

除了以上四个主要步骤，AES 还包括三种不同规格的块：128 位（AES-128）、192 位（AES-192）和 256 位（AES-256）。块加密的过程被分为不同的轮，支持 AES-128 标准的设计共由 10 轮组成。

表 7.2 列出了不同应用对吞吐量的需求，如无线网和网络电话。要做的是找到满足一种甚至几种吞吐量需求的设计。

表7.2 不同应用对吞吐量的要求

应　　用	吞吐量要求	应　　用	吞吐量要求
Wi-Fi 802.11b	11 Mb/s	PAN② 802.15 TG4（低速）	250 Kb/s
Wi-Fi802.11g	54 Mb/s	PAN 802.15 TG3（高速）	55 Mb/s
Wi-Fi 802.11i/802.11n	500 Mb/s	VoIP	64 Kb/s
MAN① 802.16a	75 Mb/s	思科 PIX 防火墙路由器	370 Mb/s

① MAN：Metropolitan Area Network，城域网。

② PAN：Personal Area Network，个人区域网。

7.3.2 AES：设计和评估

初始设计首先从芯片尺寸、设计规范、运行时需求入手（见图7.1）。假定这些需求指定了采用 PLCC㊀68 封装标准、24.2mm×24.2mm 大小的芯片。

任务是选择一个既能完成所需功能又满足面积约束的处理器。这里选择ARM7TDMI，一个32位的RISC处理器。该处理器的芯片规格有两种：一种是采用180nm工艺，尺寸是0.59mm^2；另一种采用90nm工艺，尺寸是0.18mm^2。很明显。两种处理器都能够满足 PLCC68 封装标准的初始面积需求。SimpleScalar模拟器工具集对执行 AES 所开销的时钟周期数统计结果是16 511个时钟周期，所以，对于180nm工艺、主频115MHz（根据厂商的宣传）的处理器，吞吐量是[(115×32)/16511]Kb/s＝222.9Kb/s；对于90nm工艺、主频236MHz的处理器，吞吐量是457.4 Kb/s。因此，180nm的ARM7处理器很可能只够满足表7.2中的VoIP应用，而90nm的ARM7处理器除了能满足VoIP应用还可以支持PAN应用和802.15TG4应用。

接下来将在不与最初面积约束冲突的基础上对该SoC芯片进行优化，以提高整个系统的吞吐量。下面将应用第4章中用到的技术来改变缓存大小，如果SimpleScalar工具集[30]的某种软件模型对这种应用有效，那么还将使用该工具集来评估修改后的效果。

使用SimpleScalar模拟器模拟AES软件模型，测试将一个512组的直接映射一级指令缓存的块大小从32字节扩大到64字节后产生的影响；AES的执行时钟周期数由16 511降到16 094，即降低了2.6%。假设具备基本配置不含缓存的处理器初始面积是60K rbe，一级指令缓存（L1 Cache）面积为8K rbe。如果将缓存容量加倍，总面积将由68K rbe变为76K rbe。总面积增长比率为11%，但跟2.6%的速率提高比相比，性价比太低。

ARM7已经是一款流水线指令处理器。如表7.3所示，其他架构类型，如并行流水数据通路，有很大的潜力满足需求。这些FPGA方案可以满足表7.2中所有应

㊀ PLCC：Plastic Leaded Chip Carrier，塑料引线芯片载体。

用的吞吐量需求，代价是比 ASIC[146]更大的面积和功耗。另外一种选择是，本书第 6 章提到的通过自定义指令[28]扩展处理器指令集，这里指 AES 的自定义指令。

表 7.3 Xilinx Virtex XCV-1000 上的性能和面积权衡

	基本迭代	轮内部流水	全流水
最大时钟频率/MHz	47	80	95
加密/解密吞吐量（128 位）/(Mb/s)	521	888	11300
面积［片（slice）数量］	1228	2398	12600
面积（BRAM 数量）	18	18	80
片（slice）和 BRAM 使用率	10%和 56%	19%和 56%	103%和 250%

注：全流水设计所需要的资源比 XCV-1000 所能提供的更多。

需要注意的一点是，以上讨论意在举例说明在给定的一系列条件下能得出什么样的结论。在具体实践中，还要考虑其他各种因素，如从测试集的结果或者基于 SimpleScalar 模拟器工具集的应用场景等得出的数据究竟有多大代表性。无论什么情况，这些分析只用来做出评估，它是仅用来指明可行解决方案方向的粗略性评估。在使用适当的设计工具对设计进行细化时，这些评估会得到进一步确认。

另外，有两点更深入的思考。首先，如图 7.4 所示，一个 AES 的设计可以采用流水线技术和 FPGA 芯片来实现。为了实现高于 21Gb/s 的吞吐量，具体实现时会在 FPGA 中开发专用技术架构，如块内存和块乘法[107]。

第二点，AES 的核心经常被用作大型系统的一部分。如图 7.5 所示，AES 核

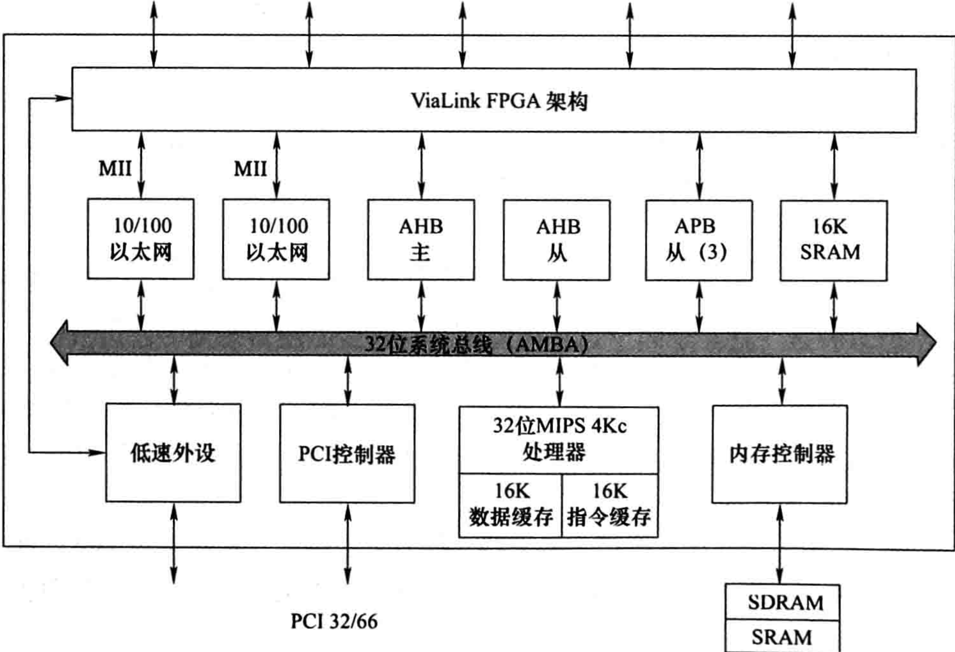


图 7.5 AES SoC 系统的 QuickMIPS 架构框图[116]

心在 QuickMIPS 设备的 ViaLink FPGA 架构[116]。这个设备包含了一个独立的 32 位 MIPS 4Kc 处理器核心和很多存储和接口单元。另一个例子就是应用 AES 的涉及安全哈希方法的设计[88]。

7.4 应用研究：三维图形处理器

本章节讨论三维图形加速器，与索尼 PS2 架构类似[237]。研究阐明了在得出初始设计的过程中有以下两个有用的技术：

- 分析　应用被看做一个比较高层次的算法，此算法用来粗略估计如计算量和通信量的数据，以便形成一个初始的设计风格和设计组件。
- 原型　利用通用的软件工具和现成的硬件平台开发出应用的一个简单版本，如运用标准的个人计算机或者通用的 FPGA 平台。在开发过程中，将会遇到需要注意的一些地方，如性能瓶颈；同时，它还将帮助鉴别出不需要进行优化的组件，从而节省开发时间。

ESL 技术和工具[32,108]通常可以帮助完成分析和原型过程。

7.4.1 分析：处理

在三维图形学中，所有物体均用三维空间中的三角形集合来表示，并且带有影响图像基本元素（像素），以及使物体更加真实的亮度和纹理信息。这些三维信息被转化到二维空间用来观看。动画包括了一系列在时间上连续的像素帧。

图形处理流水线（见图 7.6）中主要有两步：变换和光照，光栅化。下面将陆续介绍它们。

需求　对于变换和光照处理，假设一帧包含了 v 个可见的三角形，l 个光源；v 和 l 越大，物体的真实度和复杂度越高。为了方便理解透视投影，采用四维坐标系统来表示：三维空间信息和一维表示三维图像投影平面位置的信息。

在变换和光照处理过程中，每个三角形顶点都从世界坐标系转换到观察坐标系，这个过程需要计算一次 4×4 的矩阵乘法；然后将它们投影到二维平面，需要一次除法和透视纠正的四次乘法。对于每一个顶点，这个过程可以被近似为 24 次浮点乘法累加（Floatingpoint Multiply And Accumulate，FMAC），浮点除法被近似当做四次 FMAC。光照处理额外需要 4×4 次乘法来旋转顶点，一次点乘和一些进一步的计算。对于每一个光照处理需要近似为 20 次 FMAC。所以，对于每一个顶点，需要（$24+20l$）次 FMAC。在最坏情况下，每个三角形都有三个不同的顶点，每一帧需要 v（$72+60l$）次 FMAC。在最好情况下，相邻三角形的顶点可以共享，每一帧需要 v（$24+20l$）次 FMAC。

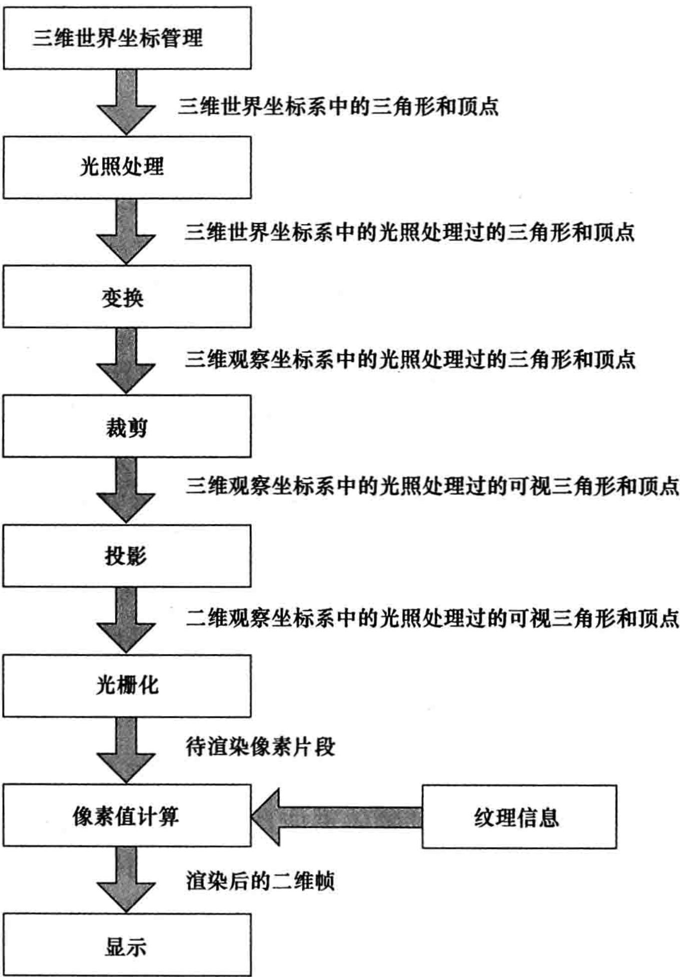


图 7.6　三维图形处理流水线

假设每秒能处理的三角形为 n 个，每秒能完成 FMAC 计算的次数为 m 次。若每秒的帧数为 f，$n = fv$，且

$$n \times (24 + 20l) \leqslant m \leqslant n \times (72 + 60l) \qquad (7.1)$$

若 $n = 50\text{M}$（$1\text{M} = 10^6$），即每秒 50M 个三角形，且没有光源（$l = 0$），那么 $1200\text{M} \leqslant m \leqslant 3600\text{M}$；若 $n = 30 \times 10^6$，即每秒 30M 个三角形，有一个光源（$l = 1$），那么 $1320\text{M} \leqslant m \leqslant 3960\text{M}$。

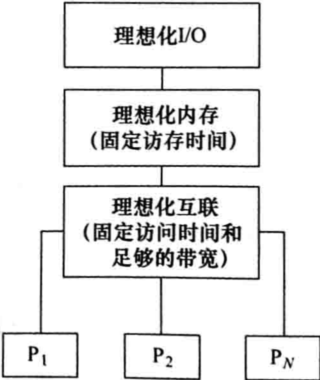


图 7.7　理想化 SoC 架构

设计　下面将描述怎么得到能满足需求的设计。初始设计是以图 7.7 所示的简单结构为基础的，这个结构在本书第 1 章已做了介绍。

提出的设计方案是从情感引擎（emotion engine）中获得灵感的，情感引擎包含了两组处理器，如图7.8所示。第一组处理器包括以下几部分：

- CPU　一个双发射、128位、32个寄存器的MIPS-III处理器。
- 浮点处理单元（Floating Point Unit，FPU）　支持基本的FMAC和浮点除法。
- VPU0　矢量处理单元，可以作为MIPS处理器的协处理器，也可以作为独立的SIMD/VLIW处理器。
- IPU　图像处理单元，解码压缩过的视频流。

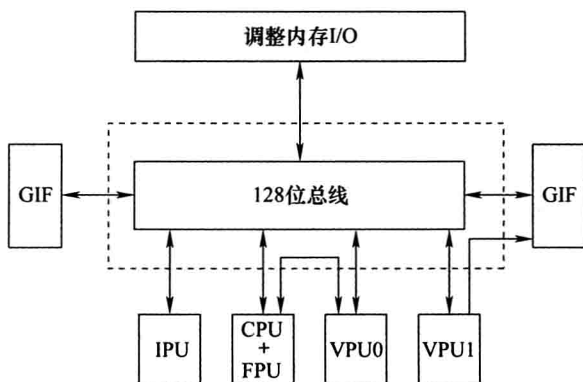


图7.8　三维图形引擎的原始设计

这些组件使用150MHz、128位的总线相连接。

第二组处理器包括以下几部分：

- VPU1　与VPU0一样，但只作为SIMD/VLIW处理器工作。
- GIF　使用128位总线将数据分流到图像合成器的图形接口。

由于VPU0和VPU1都包含了四个300MHz浮点乘法器，那么它们的性能应该为300MHz × 8 = 2400M FMAC/s；这个值是符合前面得出的1200M ≤ m ≤ 3960M。

其他的组件这里不详细进行描述。例如IPU，它是解码压缩过的视频流的图像处理单元。

需求　光栅化处理需要将每个二维三角形进行扫描转换，计算出每个三角形对应的输出像素。这个过程通常是使用数字微分分析器（Digital Differentid Analyzer，DDA）或者另一个画线（line-drawing）方法。DDA方法是沿着三角形的边垂直逐步进行的，从而使每个水平区间能同时处理。在每一个区间内，需要使用更多的DDA在适当的位置插入Z值［用作阻塞测试（occlusion testing)］、颜色和纹理信息。这里忽略了顶点的DDA，把每个像素的这个处理过程近似为需要（$2+2t$）次DDA操作。

设计　若每次DDA操作需要四条整数指令，每个像素的其他操作（如比较更新帧和z缓冲器）也需要四条整数指令，那么对于每一个处理后的像素需要的指令条数为

$$4+8(1+t) \tag{7.2}$$

每当一个像素被渲染，就要进行以上步骤，即使这个像素已经被渲染过了。假设

每帧输出 o 个像素，由于重叠的原因，每个输出像素需要重复计算 p 次，那么每帧需要的指令条数为

$$(12+8t)\times o\times p \tag{7.3}$$

在下面的原型处理中将会用到以上结果。注意这里已经忽略了顶点 DDA 计算的时间，所以期望有额外的跟 v 相关的计算时间，但整体的计算时间主要是由每个像素的处理时间组成的。

7.4.2 分析：互联

三维图形处理流水线有两个主要的任务间的逻辑通信通道：从建立/管理任务传递到变换任务的三维三角形列表；从变换任务传递到渲染任务的二维三角形列表。这两个通道本质上是无向的：一旦一个任务将一个三角形集合（二维或三维）传递到下一个阶段，除了如状态指示标志的明显的数据流信号，在通道相反的方向上没有其他的实质的数据流。

需求 在第一个通道（即介于世界管理和变换之间的通道）中，所有的三维坐标由三个单精度浮点数组成，需要 4 字节 ×3 = 12 字节。最小的三角形，即每个三角形仅包含三个坐标，需要 3 × 12 字节 = 36 字节。然而，大多数情况下，还需要额外的信息，如纹理信息和亮度信息。为了支持亮度信息，每个顶点足以存储曲面法线的。这就代表了无论光源有几个的，每个顶点的大小均为 $3\times(12+12\times\min(l,1))$。对于每个顶点，纹理信息需要额外的二维纹理坐标存储空间；假设纹理坐标是浮点数据类型，每个顶点每种纹理信息需要 8 字节。对于 n 个可见的三角形，此通道的总带宽为

$$3n\times(12+12\times\min(l,1)+8t) \tag{7.4}$$

例如，假设 $n=50\times10^6$，$l=0$，$t=1$，带宽需求则为 3GB/s；假设 $n=30\times10^6$，$l=1$，$t=1$，则带宽需求则为 2.88GB/s。

在第二个通道中，每个点处于屏幕坐标系中，所以每个点以两个 16 位整数表示的。除此以外，每个像素还需要深度信息，而深度信息是以更大的精度来存储的，所以每个顶点的总大小为 4 字节。假定使用了亮度信息，则顶点也需要颜色强度信息，即每个通道 1 字节，或者每个顶点近似 4 字节：$4+4\min(l,1)$。在光栅化时，每个纹理信息独立应用，所以需要保留 4 字节的坐标信息。所以产生的第二通道的总带宽需求为

$$3n\times(4+4\times\min(l,1)+4t) \tag{7.5}$$

这样，假设 $n=50\times10^6$，$l=0$，$t=1$，带宽需求则为 1.2GB/s；假设 $n=30\times10^6$，$l=1$，$t=1$，则带宽需求则为 1.08GB/s。

设计 如图 7.8 所示，互联总线为 128 位。由于 128 位工作在 150MHz 的总线的最高传输速率为 2.4GB/s，这只满足了第二个通道的带宽需求而不能满足第

一个，因此需要增加一个额外的专用的接口供渲染引擎使用。

7.4.3 原型技术

三维渲染器原型是使用C语言编写的。此原型融合了世界坐标管理阶段、变换阶段和基于z缓冲器的渲染器。世界坐标管理阶段创建三角形的随机模式；变换阶段将三角形投影到二维三精度浮点平面；z缓冲器渲染器采用了整数DDA。渲染器的参数可以分为以下几种：

- 三角形的个数；
- 三角形的大小；
- 输出的高度和宽度。

通过调整以上参数，可以选择性地调整如o、v和p等参数。例如，通过增加三角形的大小可以调整参数p，因为这样可以增加每个三角形与其他三角形重叠的机会。

图7.9给出了当输出像素个数增加时Athlon 1200上各阶段执行时间的变化。正如预期所示，在创建阶段和变换阶段，随着输出像素个数的增加，执行时间没有明显的变化。相反，在渲染阶段执行时间随着输出像素的个数线性增长。拟合直线具有很好的线性相关系数：0.9895。光栅化时间和像素个数之间的拟合线性关系由式（7.3）给出：$t=o\times5\times10^{-8}+0.0372$。0.0372的大偏移是由在分析这一节提出的每个三角形的建立时间引起的。基于此原型，可以感觉到这种影响是合适的，因为它们比预期的更有意义。

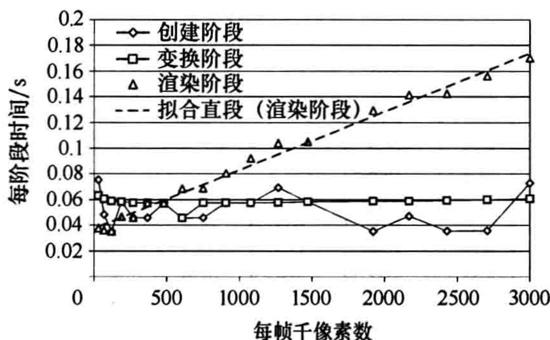


图7.9 当输出像素个数增加时Athlon1200上各阶段执行时间的变化

根据式（7.2），在没有纹理信息（$t=0$）的条件下，每个输出像素的指令数为$p\times12$。在本次实验中$p=1.3$，所以每帧的指令数为15.6。Athlon1200的官方性能为1400MIPS（Million Instructions Per Second，百万条指令每秒），依据此

模型，每个额外的像素需要 $(\frac{15.6}{1.4}\times10^{-9})\mathrm{s}=1.1\times10^{-8}\mathrm{s}$。对比预测的 $1.1\times10^{-8}\mathrm{s}$ 和实际观测到的 $5\times10^{-8}\mathrm{s}$，可以看出后者几乎是前者的5倍。产生这个错误的主要原因是原型的未优化本质和模型中的近似处理。

图7.10和图7.11给出了变换阶段在不同FPU数量条件下的性能，此测试使用PISA模拟器[30]完成。当使用四个乘法器时，性能得到了显著提高，即使使用超过4个乘法器只带来了很小的收益。这就表示一个能同时处理四个浮点运算

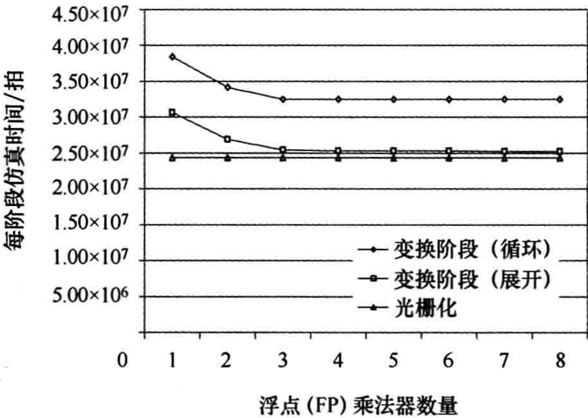


图7.10　对比初始版本和完全展开版本，不同数量浮点乘法器条件下图形变换阶段的执行时间变化（队列长度、发射宽度和提交宽度都固定为相对较大的恒定值）

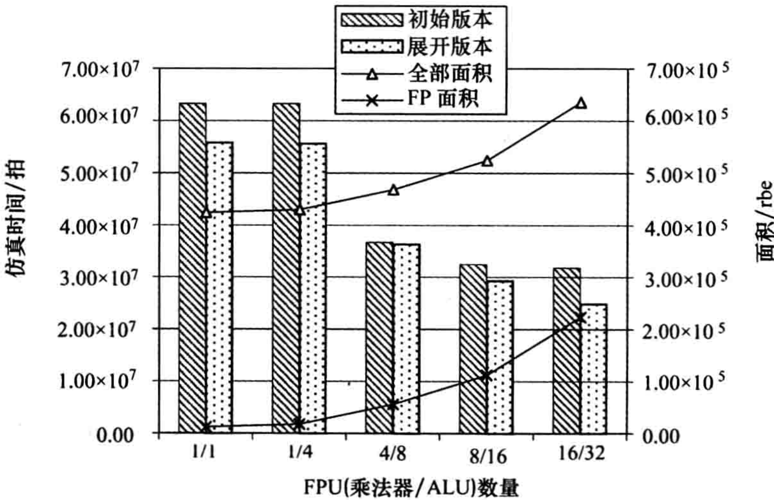


图7.11　对比初始版本和完全展开版本，不同数量FPU（乘法器/ALU）条件下图形变换阶段的执行时间变化队列长度、发射宽度和提交宽度随着FPU数量而变化。图中也给出了处理器的全部近似面积和FPU的近似面积，单位为rbe

的 VLIW 或者 SIMD 处理器将会是高效的。图中也给出矩阵乘法循环展开的情况下的性能，而不是使用双重嵌套循环来实现。展开循环更好地利用了处理器的FPU，但是所得到的加速比需要的开销也大。最后，图 7.12 给出展开变换阶段不同数量浮点乘法器和 ALU 面积和性能增长的对比 8 个浮点乘法器和 16 个 ALU情况具有单位面积的最高性能。

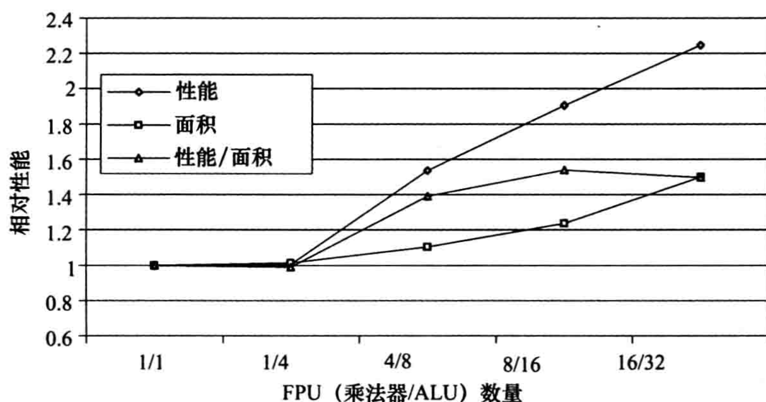


图 7.12 展开变换阶段不同数量浮点乘法器和 ALU 在面积和性能增长的对比（最大性能面积比所对应的配置为 8 个浮点乘法器和 16 个 ALU）

本节论证了应用模型在具有大量计算特征的三维引擎应用中提供了非常有用的预测功能。尽管由于估计指令数的原因导致了预测时间不准确，但是它们表现出来的整体趋势依然可以作为未来开发的依据。这个简单分析唯一没有预测的是随着三角形个数的增加光栅化时间的增长趋势。

7.5 应用研究：图像压缩

帧内操作在如 JPEG 的静态图像压缩方法和如 MPEG、H.264 等视频压缩方法中都很常见。它们包括色彩空间转换和熵编码（Entropy Coding，EC）。视频压缩方法通常还包含了帧间操作，如利用连续视频帧的相似性的运动补偿（Motion Compensation，MC）。这将在 7.6 节进行说明。

7.5.1 JPEG 压缩

JPEG 压缩方法对每个像素使用了 24 位，分别对应红、绿和蓝（Red Green and Blue，RGB）各 8 位。它可以实现有损和无损压缩。压缩过程主要有以下三步[42]：

第一步，色彩空间转换。图像从 RGB 色彩空间转换到其他的色彩空间，如

YCbCr。Y 分量代表了像素的亮度，Cb 和 Cr 分量代表了色度或色彩。相比 Cb 和 Cr 分量，人眼睛对 Y 分量更加敏感，所以对 Cb 和 Cr 分量采用了缩减像素采样。JPEG 中缩减像素采样比可以为 4∶4∶4（无缩减）、4∶2∶2（水平方向减半）和常见的 4∶2∶0（水平和垂直方向均减半）。对于压缩处理的后半部分，Y、Cb 和 Cr 采用相似的方式分别单独处理。这三个分量组成了图 7.13 所示的输入部分。

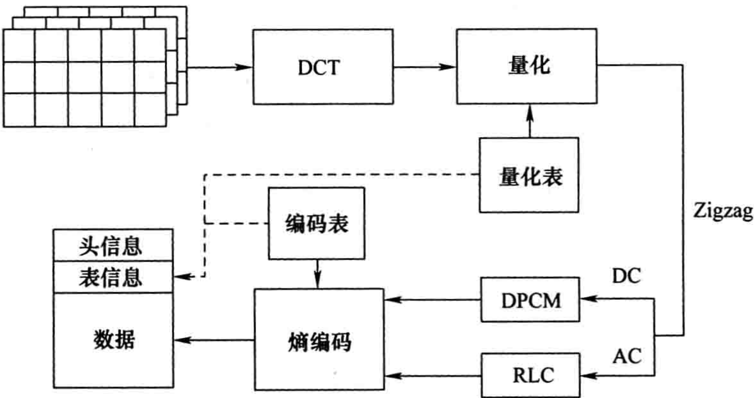


图 7.13 JPEG 压缩的模块图。其中色彩空间转换没有给出

第二步，离散余弦变换（Discrete Cosine Transform，DCT），如图 7.13 所示。图像像素的每个分量（Y，Cb，Cr）被划分为 8×8 的块，使用二维正向离散余弦变换（DCT，类型Ⅱ）将每块乘以 8×8 的矩阵，从而将此块转换到频域中。由于低频像素代表了图像的大部分信息，所以可以使用量化（另一个矩阵运算）来减少高频的信息。

第三步，熵编码。熵编码是无损数据压缩的一种特殊方法。首先将图像的数据按“Zigzag”顺序进行排序，从而使低频分量排在前面；然后使用行程编码（Run-Length Coding，RLC）算法将相近频率的数据集合到交流分量，使用差分脉冲编码调制（Differential Pulse Code Modulation，DPCM）将相近频率的数据集合到直流分量；最后使用哈夫曼编码或者算术编码处理以上得出的数据。尽管算术编码能得出更好的结果，但是编码过程也很复杂。

为了估算操作量，以二维 $k \times k$ 离散余弦变换为例，需要计算：

$$y_i = \sum_{0<j\leqslant k} c_{i,j} x_j$$

式中，$0<i\leqslant k$；x 为输入；c 为离散余弦变换系数；y 为输出。这需要完成 k 次图像数据装载，k 次系数数据装载，k 次乘累加，以及一次数据存储。所以，对于每个像素，总计需要 $3k+1$ 次操作。因此，对于 $k \times k$ 块的行列分解的 DCT，需要 $2k^2(3k+1)$ 次操作。

对于帧率为f/s的 $n\times n$ 分辨率的情况，所需要的操作次数为 $2fn(3k+1)$。两种常见的格式为通用中间格式（Common Intermediate Format，CIF）和四分之一通用中间格式（Quarter CIF，QCIF），分别对应为 352×288 像素和 176×144 像素的分辨率。

对于色彩空间为YCbCr，帧格式为QCIF，采样率为4:2:0，有594个 8×8 的块，帧率为15f/s的情况，所需要的总操作次数为 $2\times15\times594\times8\times8\times(24+1)=28.5$MOPS⊖。对于CIF帧格式，需要114MOPS。

通常，无损压缩能达到三倍的压缩效果，而有损压缩可以高达25倍的压缩效果。

7.5.2 实例：数字静态相机中的JPEG系统

数字静态相机（digital still camera）的一个典型的图像处理流水线如图7.14[128]所示。TMS320C549处理器接收到SDRAM的 16×16 像素块后开始进入图像处理流水线。

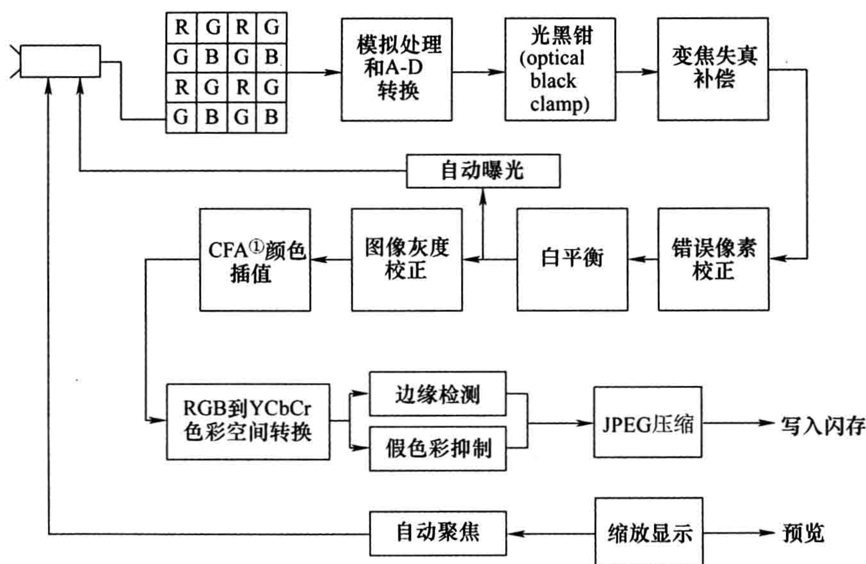


图7.14 静态相机[128]的模块图

① CFA：Color Filter Array，彩色滤波阵列。

由于TMS320C549处理器拥有一个32K的16位RAM和一个16K的16位的

⊖ MOPS：Million Operations Per Second，每秒百万次操作。

ROM，而 16×16 像素块所需要的存储空间很少，所以所有的图像处理流水线操作都可以在片上执行。这样，处理时间将会保持很短，因为不需要进行延迟较长的外部访存操作。

此处理器的性能高达 100MIPS，且功耗较低，仅有 0.45mA/MIPS。表 7.4 给出了 TMS320C54X 处理器的性能，包括了图像处理流水线程序各个不同阶段所需要的精确的时钟拍数。对于每个像素，包含了整个 JPEG 的图像处理流水线需要大约 150 拍，或者相当于一个 100MIPS、100MHz 的处理器执行了 150 条指令。

表 7.4 TMS320C54X 的性能[128]

任务	拍/像素
预处理，如捕获、白平衡	22
色彩空间转换	10
插值	41
边缘增强，伪彩色抑制	27
4：1：1 抽取，JPEG 编码	62
共计	162

一个工作在 100MHz 的 TMS320C54x 处理器能够在 1.5s 内处理 1 张百万像素电荷耦合器件（Charge Coupled Device，CCD）图像。这个处理器支持 2s 的拍摄延迟，其中包含了将数据从外部存储传输到片上存储。数字相机同样必须支持用户在 LCD 屏幕上或者是在一个外部的电视显示器上显示拍摄的照片。由于拍摄的照片存储在闪存上，所以 SoC 还需要一个回显程序。

假如图像是以 JPEG 比特流的方式存储，回显程序将它们进行解码，并缩放到合适的分辨率，然后将它们显示在 LCD 屏幕或者外部电视显示器上。工作在 TMS320C54x 处理器上的回显程序能够在 100 拍内处理一个像素，即支持了百万级像素图像的 1s 回显速度。

此处理器需要 1.7KB 的指令存储和 4.6KB 的数据存储来支持图像处理流水线和 JPEG 标准的图像压缩。完整的图像处理流水线程序存储在片上，这就减少了外部访存的次数，从而使应用慢速外存成为可能。这种结构不仅能提高性能，而且能降低成本和提高能效。

近年来，数码相机中的芯片需要额外支持图像压缩、视频压缩、音频处理和无线通信[217]。图 7.15 给出带视频、音频和网络处理能力的相机芯片模块图，包括了这种芯片的几个关键部分。

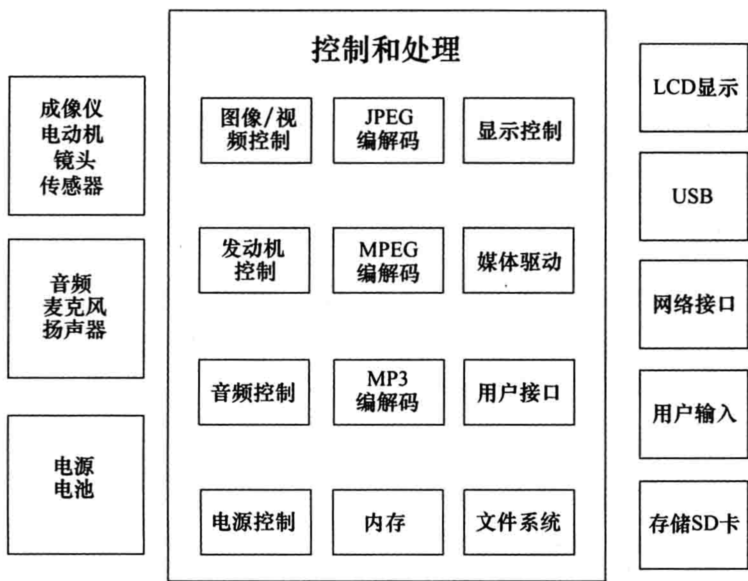


图7.15 带视频、音频和网络处理能力的相机芯片模块图

7.6 应用研究：视频压缩

表7.5给出了一些常见的视频格式，包括了各种不同应用中的常用视频格式及相关的压缩方法，如MPEG1、MPEG2和MPEG4。视频质量是由比特率和视频分辨率决定的，比特率越高，视频分辨率越高，就代表了视频质量越好，但同时意味着更高的带宽需求。

表7.5 一些常见的视频格式

格 式	VCD	SVCD	DVD	HDDVD HDTV (WMVHD)	AVI DivX XviD WMV	MOV Quick-Time
分辨率	352×240	480×480	720×480 *	1920×1080 *	640×480 *	640×480 *
NTSC/PAL	352×288	480×576	720×576 *	1280×720 *		
视频压缩方法	MPEG1	MPEG2	MPEG2, MPEG1	MPEG2 (WMV-MPEG4)	MPEG4	MPEG4 (源于 Sorenson Media)
视频比特率	1150 Kb/s	2000 Kb/s†	5000 Kb/s†	20Mb/s† (8Mb/s†)	1000 Kb/s†	1000 Kb/s†
每分钟数据量	10 MB/min	10～20 MB/min	30～70 MB/min	150 MB/min† (60MB/min†)	4～10 MB/min	4～20 MB/min

(源于：http：//www.videohelp.com/svcd)

* 近似分辨率，可以高于也可以低于此值。

† 近似比特率，可以高于也可以低于此值。

另一组压缩方法有 H. 261、H. 262、H. 263 和 H. 264，其中有一些与 MPEG 方法是相关的。例如，MPEG2 与 H. 262 是一样的，H. 264 与 MPEG4/Part10 是一样的。通常越新的方法，如 H. 264，提供了越高的视频质量和越高的压缩比。下面大致介绍一下这些视频压缩方法[42,270]，但不做详细的描述。

7.6.1 MPEG 和 H. 26X 视频压缩：需求

除了前面描述的静态图像的帧内压缩方法，视频压缩方法通常还包括了如运动估计等帧间压缩方法。在标准视频编码里，运动估计具有最多的操作次数，如下面显示了 H. 261 压缩方法对单帧为 CIF 格式、352 ×288 像素的帧率为 30f/s 的视频流的需求：

压缩需要 968MOPS，则有

RGB 转换为 YCbCr	27
运动估计	608（在 16 × 16 的区域内进行 25 次搜索）
帧内/帧间编码	40
循环滤波	55
像素预测	18
二维 DCT	60
量化和 zigzag 扫描	44
熵编码	17
帧重建	99

大多数压缩标准的计算量都是不对称的，即解压比压缩容易。例如 H. 261，解压的操作量（大约 200MOPS）大约是压缩（1000MOPS）的 20%。

解压需要 198MOPS，则有

熵解码	17	循环滤波	55
反量化	9	预测	30
IDCT⊖	60	YCbCr 转换为 RGB	27

MPEG 运动估计方法涉及三种帧（见图 7. 16）。第一种，I 帧，它不包括运动信息，所以类似有损 JPEG 格式。第二种，P 帧，它包含了基于前面的 I 帧的运动预测信息；它也包含了运动矢量（Motion Vector，MV）和误差项。由于误差项都较小，所以量化会得到很好的压缩比。第三种，B 帧，它是双向帧，同时

⊖ IDCT：Inverse DCT，反离散余弦变换。

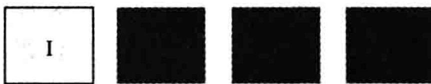
包含了基于前面和后面的I帧或者P帧的运动预测信息。

运动估计的目的就是利用参考帧的信息来描述当前帧的信息。参考帧通常就是编解码顺序中当前帧的前一帧的重建版本。

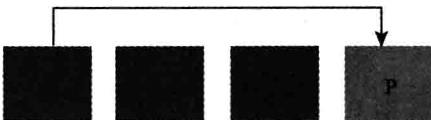
H.264等视频压缩方法的运动估计器中，这些成对的帧对像素值进行处理。基于运动估计器的操作，当前帧的像素值交替使用一组两个值进行表示：基于参考帧的预测像素值加上预测误差。预测误差代表着当前帧的预测像素值与真实像素值之间的差别。

运动估计器的功能可以用以下方式来解释。假如原始视频序列中的一个运动物体在当前帧中以一组像素来表示，同时它也以重建顺序中的前一帧的一组像素来表示。重建顺序决定了参考帧。为了达到压缩效果，当前帧中物体的代表像素是以参考帧中的像素值推理出来的。参考帧中代表物体的像素值被称为预测器，因为它预测了当前帧中的物体的像素值。预测器通常需要一些变化来获取当前帧中真实像素值，这些变化被称为预测误差。

• I帧作为静态图像编码，不依赖任何参考帧



• P帧依赖前面显示的参考帧



• B帧依赖前面和后面的参考帧

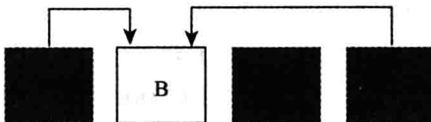
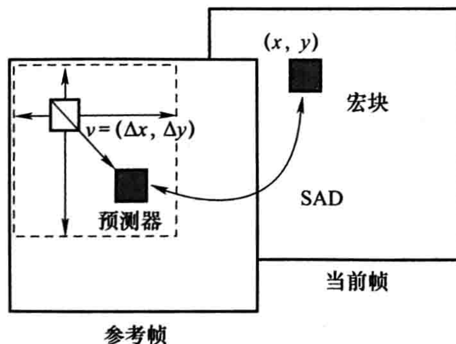


图7.16 MPEG运动估计中的三种帧

在基于块的运动估计中，当前帧中的物体边界被假定为沿着宏块的边界对齐。基于此假设，帧中描述的物体可以用一个或者多个宏块来表示。单个宏块内部的所有像素都具有相同的运动特征。这些运动特征用宏块的运动矢量来表示，这个矢量是从当前帧的宏块像素位置指向它的预测器的像素位置。视频编码标准没有规定运动矢量和预测误差是怎么获得的。

图7.17给出了运动估计的过程。通常，运动估计只处理帧的亮度部分，所以每个宏块对应了16×16个亮度像素。当前帧被划分为不重叠的宏块，对于每个宏



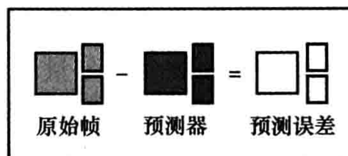


图7.17 运动估计的过程

块，运动估计器需要指定一个同样也是一个 16×16 的方形预测器。预测器同样也可以被看做是一个宏块，预测器依据是它与当前帧中的宏块的相似度来选择。

H. 264 等视频编码标准没有规定宏块相似度的衡量标准，一个很常见的标准是绝对距离的总和（SAD）标准，SAD 方法就是计算两个宏块中相对应亮度像素的 SAD 值。

参考帧搜索区域内的所有宏块都要使用 SAD 方法计算出 SAD 值。SAD 值越大代表着这两个宏块的区别越大。最小 SAD 值的宏块将作为预测器宏块。

很多视频压缩标准都要求搜索区域为矩形且搜索区域必须位于原始宏块的坐标系上。矩形的维度是可调节的，但是不能超过标准规定的最大值。

运动搜索就是寻找最佳匹配的预测器宏块，视频压缩标准通常也不会规定运动搜索策略。当前已经有很多运动搜索方法被提出来了。其中一种可能的搜索方法就是穷尽（或全部）搜索，即对搜索区域内的所有可能的宏块都进行搜索，这种策略能够保证得到的 SAD 值是搜索区域内全局最小的。但是，穷尽搜索的计算代价非常大，穷尽搜索由于其规则性，所以主要会被硬件设计者采用。

图 7.18 给出了不同视频压缩方法的带宽和存储，是不同压缩方法压缩 90minDVD 质量的视频所需要的带宽大小和存储大小。

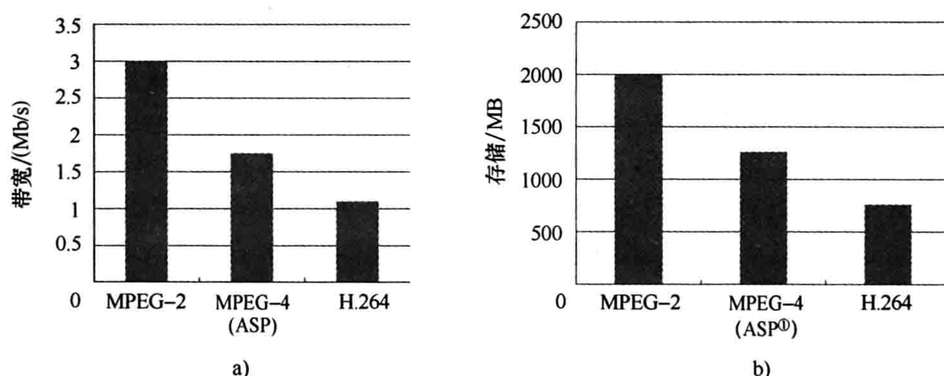


图 7.18 不同视频压缩方法的带宽和存储

a）带宽 b）存储

① ASP：Active Simple Profile，主动采样技术，是 MPEG-4 的一个版本。

观察压缩或者解压算法各个操作的执行时间百分比时，可以发现运动补偿和运动估计通常占据最多的分量。例如，H. 264/AVC（高级视频编码）解压算法包含了四个主要步骤：运动补偿、整数转换、熵编码和去块滤波。如图 7.19 所示，其中运动补偿需要开销最多的时间。

压缩算法也有相同的结论。例如 H. 263 软件编码[270]中运动估计占了约 95% 的执行时间（见图 7.20）。

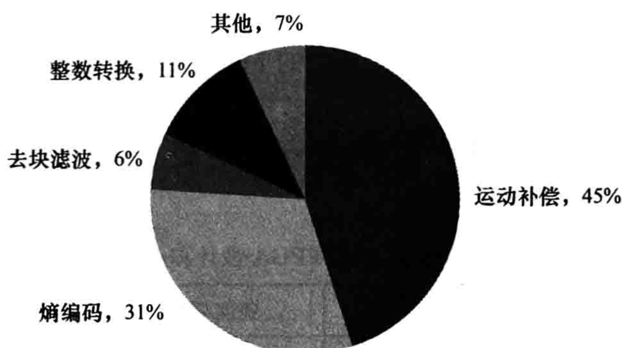


图 7.19 H.264/AVC 解压的核心阶段对比

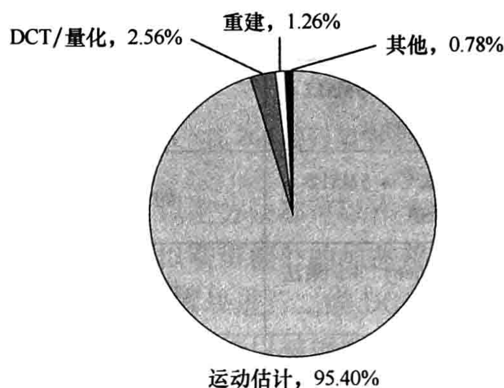


图 7.20 采用半像素穷尽运动搜索[270]的基本编码方式的 H.263 编码核心阶段的对比

7.6.2 H.264 加速：设计

H.264 是最常见的视频格式之一。它作为一个国际标准，已经被很多 DVB 和 3G 等广播和移动通信标准所采用。它的编码效率使视频流等新的应用成为可能。

H.264 包含了视频压缩中的运动估计和其他方法等高级算法。对于运动估计，它支持 16×16 个像素到 4×4 个像素的宏块大小。对 4×4 的宏块使用残留数据转换来消除舍入误差，残留数据转换的方法为改进的整数 DCT。

相比以前的标准，H.264 具有更好的性能，原因在于它的搜索范围更宽，具备多参考帧，并且运动估计和运动补偿采用更小的宏块（这将导致计算中更多的装载次数）。为了满足多种不同的编码任务，如运动估计中的各项操作，H.264 需要更高速度的内存和高度流水化的设计。

现在考虑两种方式。第一种方式是以可编程或者专用硬件来实现这些任务。例如，有一项最新的设计以不同的硬件技术实现了基本的 H.264/AVC 编码核

心[158]，包括以下几项（见表7.6）：

- 4CIF（704×576），30f/s；低成本的FPGA，Xilinx Spartan-3 和 Altera Cyclone-Ⅱ。
- 720P，30f/s；高端FPGA，Xilinx Virtex-4 和 Altera Stratix-Ⅱ。
- 1080P，30f/s；0.13μm ASIC。

表7.6 H.264/AVC编码的FPGA设计和ASIC设计[158]

技 术	近似面积	速度/MHz	视频吞吐量
0.13μm LV 0.9V，125C	178K 门 + 106Kb RAM，为速度做了优化	约250	1920×1080（1080P）帧率为30f/s
0.18μm 低速工艺	129K 门 + 106Kb RAM，为速度做了优化	约50	4CIF（704×576）帧率为30f/s
StratixIIC3	17511 ALUT① +5M512 + 51M4K +3DSP	约118	1280×720（720P）帧率为32f/s
CycloneIIC6	18510M4K② + 5M512 + 51M4K +3DSP	约65	4CIF（704×576）帧率为40f/s
Virtex4-12	10600片（Slice） +3乘法器+33RAM块	约110	1280×720（720P）帧率为30f/s
Spartan3-4	10600片（Slice） +3乘法器+33RAM块	约50	4CIF（704×576）帧率为30f/s

① ALUT：Adaptive Lookup Table，自适应查找表。
② M4K：一种可配置内存块，总大小为4608bit。

第二种方式是以本书6.5.3节介绍的软件可配置处理器来实现这些任务，此处理器具有指令集可扩展结构，用来支持实现定制需求的操作，即定制指令（也称为扩展指令）。这里使用的是一个300MHz的Stretch S5处理器，它能达到30f/s对标清（Standard Definition，SD）格式的视频流进行H.264进行编码[152]。下面将深入了解此方法。

一个成功的实时视频编码应用必须能够为特定的屏幕分辨率传递最好的图像质量，这将受到现实世界的限制。例如，一个未压缩的720×480像素的视频流，每个像素需要1.5B的颜色信息。这样的视频流每帧需要518KB，在30f/s的条件下，它需要15.5MB/s的存储能力和带宽。

图7.21给出了H.264编码架构。这种编码方式的效率可以从以下功能的实现中看出：

1. 前向DCT和IDCT。
2. 利用前向和反矢量化进行帧内预测。

3. 去块滤波。
4. 利用帧间对比进行运动估计。

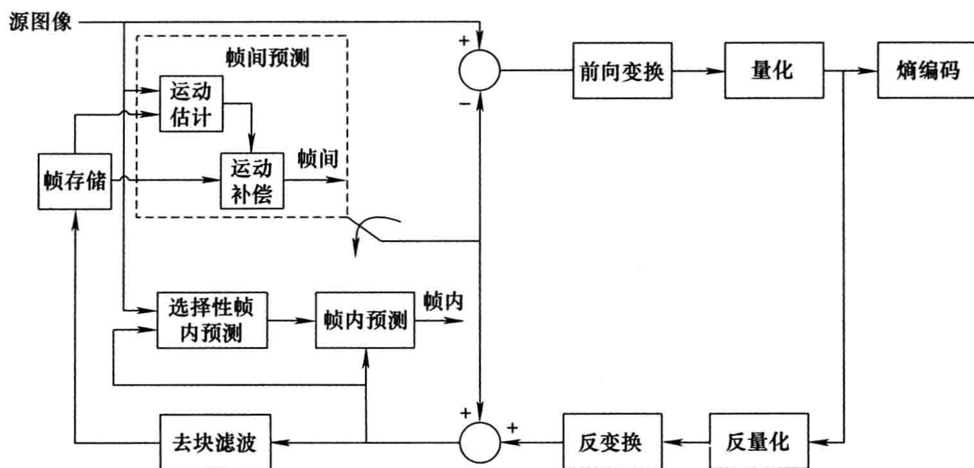


图 7.21　H.264 编码架构[152]

由于需要的计算量的限制，以上方法都可以作为硬件加速的主要备选项。通过利用这些算法的内部并行性可以获得额外的加速效果。

DCT　考虑对 4 ×4 的亮度像素块进行二维 DCT 和量化操作。利用对称性和公共子表达式，DCT 部分的矩阵计算可以被缩减为 64 次加减计算。所有的 64 次操作可以被 ISEF 中的一条定制指令来实现。

量化（Q）　这一步在 DCT 之后。将量化操作用简单的乘法和移位操作来实现，可以避免计算量较大的除法操作。将亮度像素用 DCT + Q + IDCT + IQ 进行编码和解码的过程一共需要 594 次加法、16 次乘法和 288 次选择（采用多路选择器）。

去块滤波　ISEF 的 128 位总线需要 1 拍将一行 8 个 16 位的预测数据装载进来。所以在编译器能够识别出函数的内部并行性的条件下，一条 ISEF 指令就可以替代多条传统指令。使用标准处理器对 4 ×4 的宏块进行这些处理需要超过 1000 拍，但在软件可配置的处理器中执行只需要 105 拍，相当于 10 倍的加速比。因此一个 30f/s、720 ×480 像素的视频流只需要 RISC 处理器的 14.2% 的利用率，因为任务的大部分都被卸载到 ISEF 里。增加子宏块大小可以提高并行性；例如，对两个 4 ×4 宏块并行进行处理能减少一半的执行时间，使 RISC 处理器的利用率降低到 7.1%，原因是 ISEF 承担了更重的负载。

加速去块滤波要求开发者将条件代码减到最少。不需要决定计算什么值，取而代之的是更高效地创建一条定制指令，定制指令用硬件计算所有的结果，然后选择出合适的结果。

将 IDCT 阶段的 128 位结果进行重新排序，从而简化了将 16 个 8 位的边缘像素数据打包为一个 128 位的数据包并提供给去块定制指令的过程。在 ISEF 和指令中的状态寄存器能实现对宏块参数的预计算，这是另一个可选的优化方法。

滤波器的内部循环装载 128 位的寄存器值，执行去块滤波定制指令，每条指令能计算两条边。由于同样的定制指令能够用于水平和垂直方向的滤波，所以不需要额外的开销。

内部循环需要 3 拍完成，并且执行两次（水平和垂直方向），循环的额外开销大概为 20 拍。若每兆字节数据有 64 条边，那么大约需要 416（即 64/4 ×26）拍来处理这 1MB 数据。对于分辨率为 720 ×480、帧率为 30f/s 的视频流，大约需要 16.8 兆拍每秒，或大约 5.2% 的处理器利用率。

运动估计 已经知道这部分消耗了处理器的大部分资源（50% ~60%）。它的关键计算需求就是重复的计算 SAD 值来获得最匹配的运动矢量。

运动估计需要进行重复性的数据计算和比较，并且有很多重复使用的中间结果。传统处理器和 DSP 有限的寄存器空间不能满足这些大的数据集存储需求。同时，这些处理器和 DSP 设计从数据缓存中获取的数据很难满足固定计算和乘法单元的需求。

使用 Stretch S5 软件可配置处理器时，ISEF 定制处理单元能够并行处理计算任务，并且在执行全流水 SAD 指令时，能够在状态寄存器中保存中间结果。

每个宏块的运动估计包含了大约 41 次 SAD 和 41 次 MV 计算。单个宏块执行一次全运动搜索需要 262K 次操作，30f/s 的视频流每秒需要一共 10.6G 次操作。

利用实现时的启发式算法，应用开发者可以将计算量最小化，来满足目标图像质量或者比特率要求。

跨不同搜索区域、不同帧、不同运动矢量的用户运动估计算法经过优化后可以很容易用 ISEF 指令来实现。一条定制指令就可以代替多次计算，同时也可以使用中间结果将大量计算流水化。

例如，一条定制指令可以执行 64 次 SAD 计算。ISEF 保存这 64 次计算的结果，并提供给下一条指令重复利用，从而减少了数据传输的次数。ISEF 指令也可以流水化，从而提高计算能力。

运动估计同时也涉及多种像素预测，需要在像素周围的 9 个方向执行 9 次 SAD 计算。利用定制指令，一次 16 ×16 的 1/4 像素精度的 SAD 计算需要 133 拍，一次 4 ×4 的 1/4 像素精度的 SAD 计算需要 50 拍。

以上讨论均假设使用 Stretch S5 处理器。Stretch S6 处理器带有一个可编程的加速器，它为运动估计提供了一个专用的硬件模块，所以在 Stretch S6 处理器中不需要实现这些操作。

7.7 未来的应用研究

本节介绍几种应用，从而阐明需求和 SoC 解决方案的多样性。

7.7.1 MP3 音频解码

MP3（MPEG-1/2 Audio layer-3）是最流行的高质量压缩音频格式。本节将概括地了解它的基本算法[42]并介绍它的两种实现方法：一种是用 ASIC，另一种是用 FPGA[117]。

需求 MPEG-1 标准包含了以联合比特率 1.5Mb/s 的方式压缩数字视频和音频。这个标准可以分为几部分，其中第三部分主要处理音频压缩。根据不同级别的复杂度和性能，音频压缩标准包含三层；层面三标准（通常称为 MP3）具有最好的性能，同时也是最高复杂度的。

MP3 音频算法涉及感知编码，如图 7.22 所示。这个算法将心理声学模型和混合子带/变换编码原理的结合。音频信号被分为 32 个子带信号，对每个子带信号进行改进的离散余弦变换（Modified DCT，MDCT）。根据心理声学驱动的感知误差机制，变换系数使用标量量化和变长哈夫曼编码方法进行编码。

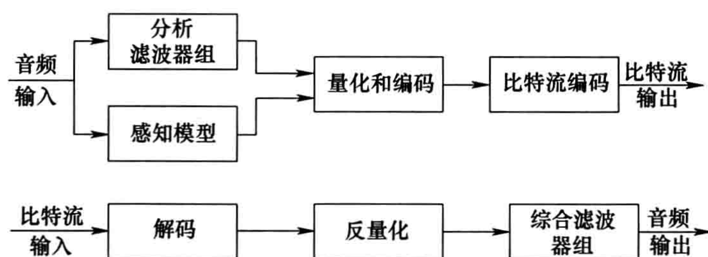


图 7.22　感知编码和解码的模块图[42]

MP3 比特流是一系列数据“帧”的串联形式，每帧对应了音频的两个“颗粒”，每个颗粒被定义为精确的 576 个连续的音频样本。1 个颗粒有时候会被划分为 3 个包含 192 个样本的短颗粒。

对一个 MP3 帧进行解码主要有三步：第一步，同步到帧的开始，解码头信息；第二步，解码边信息，边信息包括比例因子选择信息、块分割信息和表选择信息；第三步，解码两个颗粒的主要数据，包括变换因子的哈夫曼比特信息、比例因子。帧中的主要数据可能会溢出到相邻的帧，所以需要缓存多个数据帧。

当帧的信息被解析后，下一步就是从解码的信息重建音频的每个颗粒，主要包括以下几步：

1. 从主要信息和边信息中通过反量化得到变换因子，使用非线性变换来得

到解码的变换因子。

2. 当使用短块时，反量化得出的变换因子需要重拍并且划分为三个变换因子集合，每个块对应一个。

3. 当使用特定的立体信号，左声道和右声道一起进行编码时，变换因子通过声道信息重新形成左声道因子和右声道因子。

4. 对于较长的块进行“混叠消除（Alias Reduction）”。

5. 对每个声道的32个子带信息使用带因子的逆向的改进离散余弦变换（Inverse MDCT，IMDCT）。

6. 对连续帧的IMDCT输出结果使用叠加机制。明确地说，就是IMDCT输出结果的前半部分与前一个颗粒对应子带的IMDCT输出的后半部分进行叠加。

7. 最后一步，使用一组反多相滤波器将32个子带信号组成一个全带时域信号。

设计　ARM处理器和DSP处理器的结构图揭示了子带合成滤波（synthesis filter bank）是最消耗时间的任务（见表7.7）。

表7.7　ARM和DSP处理器上的MP3分析结果

模　　块	在ARM上的时间百分比	在DSP上的时间百分比
头、边声调，译码比例因子	7	13
哈夫曼编码，立体声处理	10	30
混叠消除，IMDCT	18	15
滤波器组	65	42

美国AMI半导体公司[117]已经在五个金属层350nm CMOS工艺的基础上开发出了一个ASIC原型。该芯片包含了5个RAM，有主存、哈夫曼表ROM（见表7.8），核心的大小接近13mm^2，工作在2V、12MHz时功耗为40mW。在满足实时性限制的条件下，可以将时钟频率降低到4～6MHz。

表7.8　350nm ASIC技术上的MP3解码模块[117]

解码模块	存储/bit	ROM表/bit	等价的门数量
同步	8192	0	3689
共享主存	24064	0	1028
哈夫曼编码	0	45056	10992
再量化	0	0	21583
重排	0	0	3653
反混叠	0	0	13882
IMDCT	24064	0	61931
滤波器组	26112	0	31700
I^2S	9216	0	949
共计	91648	45056	149407

解码过程的实时性要求是由 MP3 帧的音频信息决定的。解码过程中不同子模块在 24MHz 的系统时钟条件下的计算时间不同（见表 7.9）[117]。解码过程所需要的总时间为 2.3ms，在 24.7ms 的限制内显得很容易。这就表示解码的时钟速度还可以降低，资源可以共享以达到降低成本的目的。

表 7.9 Xilinx Virtex-Ⅱ1000FPGA 平台上 MP3 解码的资源利用率和计算时间[117]

解码模块	片（slice）（%）	RAM 块（%）	计算时间/μs
同步	15	10	140
哈夫曼编码	11	7	120
再量化	12	5	140
重排	1	12	10
反混叠	3	0	83
IMDCT	8	13	678
滤波器组	6	10	1160
共计	56	57	2331

对应 Virtex-Ⅱ 1000FPGA 平台上的资源利用率（见表 7.9），此设计使用了 56% 的 FPGA 片（slice），15% 的触发器，45% 的四输入查找表，57% 的块 RAM。而且，由四个可用的 18×18 位乘法器组成的 32×32 位乘法器在各个子模块中共享。

利用子模块之间的资源共享可以减少设计的面积和功耗。然而，这种资源共享可能会使控制和实现复杂化，如引入了路径瓶颈；因此，在采用这种方法前必须对它的好处与坏处进行评估。

7.7.2 IEEE 802.16 软件定义无线电

WiMAX（IEEE 802.16）无线通信标准和很多其他无线通信标准一样都试图通过提高数据传输速率来满足终端应用的需求并减少部署成本。从很多的噪声中鉴定数字数据的技术方法给大多数处理器的计算能力增加了很大的压力。随着标准的不断改进，需求的随时变动，可编程的解决方案越来越吸引人。

需求 图 7.23 显示了一个基本的 IEEE 802.16 发射器的实现模块图[169]。在较高的层面上，发射器的物理（Physical PHY）层将原始数字数据流转化为复杂数据流，复杂数据流经过进一步转化成为模拟无线电信号。接收端的 PHY 层从复杂数据流中将数据解码为原始数据形式。

PHY 层中的模块都是计算量需求较大的，包括快速傅里叶变换（Fast Fourier Transforms，FFT）和反向 FFT（IFFT）、前向纠错（Forward Error Correcti FEC）、正交调幅、交错和置乱。其中前向纠错包括如里德-所罗门编解码的分组编码，以及如卷积编码、维特比解码和正交振幅调制（Quadrature Amplitude Modulation，

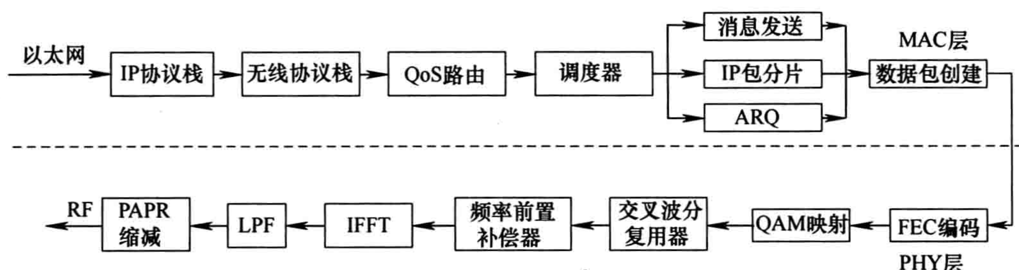


图 7.23 IEEE 802.16 发射器模块图[169]

QAM）的位编码。媒体访问控制（Media Access Control，MAC）层给 PHY 层和网络层提供接口。MAC 层的处理是控制导向的，它从网络层获取数据包，依据服务质量（Quality of Service，QoS）来对需要发送的数据进行调度；在接收端，MAC 层将数据重组并转交给网络层。MAC 层也需要在基站和用户站之间发送维持通信的信息和错误数据包的重复请求信息。网络层是面向应用的接口。TCP/IP 网络协议栈是当今最常见的网络协议栈。所有的层面相连接以组成一套完整的网络解决方案。

设计 802.16WiMAX 标准的 PHY 层需要处理 256 位的 FFT 和正交频分复用（Orthogonal Frequency-Division Multiplexing，OFDM）。在 Stretch S5 软件可配置处理器（本书 6.5.3 节）上，OFDM 的工作信道宽度可以被配置为 3.5MHz、7MHz 和 10MHz[169]，支持二进制移相键控法（Binary Phase Shift Keying，BPSK）、正交相移键控法（Quadrature Phase Shift Keying，QPSK）、16QAM 或者 64QAM 的调制方法。在带有噪声的环境里，需要 FEC 模块进行前向纠错，标准里有很多方法供选择。注意，传统的 RISC 处理器或 DSP 处理器不能满足 WiMAX 同时进行基带处理和控制任务的高级需求。单个的软件可配置处理器就可以满足责任重大的 WiMax 信号处理和任务控制的需求，如在单芯片上的基本 MAC 层和完整 TCP/IP 协议栈就能使三种信道宽度达到最高比特率。

Stretch 软件可配置处理器通过定义一条作为扩展指令的用户指令来实现基 2FFT，这条指令支持 16 次 16×16 的乘法，8 次 32 位加法和 16 次带舍入和缩放的 16 位加法。此用户指令利用 128 位位宽寄存器将三组四个复杂的值传递给 ISEF 进行并行操作。这样就能在 4μs 内完成 256 点 FFT。实现基 4FFT 能得到 28% 的性能提升。

FEC 模块是从灵活性和性能获得好处的另一个模块。在有声道噪声的情况下，前向纠错在发射端采取数据冗余，在接收端纠正错误，从而增加数据吞吐量。在卷积编码中，每一个编码位是由当前输入的 1 比特与之前输入的多个比特进行卷积运算得出的。计算中使用的比特数叫约束长度，比率为每输出 1 个比特所需要输入的比特数。WiMax 标准使用的卷积编码的约束长度为 7、比率为 1/2，

同时，它也支持其他比率。

RISC 处理器处理卷积编码中的比特级的操作时通常不够高效。在 Stretch 软件可配置处理器中，比特级的操作可以利用用户处理单元 ISEF 来进行优化。可以实现 64 位输入产生 128 位输出的用户指令，这条指令使用内部状态来保存 6 位的状态位，并将 64 位输入与状态位并行地进行卷积运算，产生 128 位的输出。

卷积编码产生的比特流可以使用格构（trellis）图来找到最可能的编码序列。维特比解码器通过限制检查的序列数目来达到高效的解码比特流。它记录每个格构阶段每个状态的最可能路径。维特比解码方法是计算密集型操作，它需要对每个阶段每个状态进行相加-比较-选择（Add-Compare-Selected，ACS）运算，同时还要记录所选路径的历史。它主要包括三步：（1）分支度量计算，（2）每个阶段每个状态的 ACS 计算，（3）回溯。

在格构图中，每个分支都有一个度量方法，这个方法称为分支度量，它衡量了接收到的信号与输出分支标签之间的距离。分支度量是由接收到的样本与分支标签之间的欧氏距离计算而来的。

用户创建的指令 EI_ ACS64 就是用来处理这些的。它进行分支度量计算，将分支度量与上一个阶段的路径度量相加，然后比较两条形成路径的路径度量，将最大值更新为新的路径度量，然后选择路径。EI_ ACS64 指令并行地处理一个格构状态的所有状态的 ACS 操作。换句话说，这条用户指令并行地处理 32 次蝶式运算。64 次路径度量的结果在 ISEF 中以内部状态存储。当一个状态转换到下一个状态时，EI_ ACS64 针对每个状态对输出宽度寄存器更新 1 位，表示此路径被选中。每个状态遍历四个格构阶段，共更新 4 位，对于所有的状态，一共更新 4 位 ×64 = 256 位，可以使用两条存储指令（WRAS128IU）将这些位存储到内存中。

真正将数据解码为原始数据是对格构沿着最可能的路径进行回溯得到的。回溯的长度一般是卷积编码的余数长度的 4 ~ 5 倍。在某些情况下，回溯开始前数据帧就已经接收完成。那么从反方向遍历格构来对输入比特流进行解码。假设在最后一个格构阶段的状态停在了一个已知的状态，典型的状态“0”。可以通过发送额外的 $K-1$ 位 0 来将所有的状态转化到“0”状态。每个状态存储的 1 位表明从阶段 j 跳转到阶段 $j-1$ 时是从哪个分支转换的。创建另一个用户指令 VITERBI_TB，用它来对 4 个格构阶段进行回溯。它使用内部状态来保存前一状态，使用它来进行下一个回溯循环，并输出 4 位的解码比特流。在 8 位解码比特流被存储到内存之前，指令 VITERBI_TB 会被执行两次。

7.8 总结

希望本章的内容能很好地阐述 SoC 应用的多样性、相关设计技术和 SoC 架构

的范畴。SoC 架构的范畴可以从嵌入式 ARM 处理器到可重构芯片，可重构芯片有从 Xilinx 到 Stretch 的各类处理器，其中有很多在前面章节中已经介绍过了。有趣的是，多媒体、密码学、通信和很多其他的关键应用对性能需求的快速增长导致了一批新兴的企业，如美国 Achronix、美国 Element CXI、荷兰 Silicon Hive、美国 Stretch 和美国 Tabula 公司，时间将会说明它们中间谁是赢家。

但是，这里没有尝试提供详细完整的使用最新工具的特定应用的设计开发或设计风格，这些内容可以在已有的书中找到。例如，Fisher 等人[93]，以及 Rowen 和 Leibson[207] 相对比较专注于 VLIW 结构和可配置处理器方面。从工业角度的 SoC 设计案例[164] 和设计方法[32] 也可以在现有的资料中找到。对处理器设计中的分析技术应用的详细案例感兴趣的读者可以参考 Flynn[96]、Hennessy 和 Patterson[118] 的书。

7.9 习题

1. 为了支持 IEEE 802.11bWi-Fi 的 AES，32 位 ARM7 指令集的处理器的频率最低需要多少？如果是 64 位处理器呢？

2. 对于 30f/s 的 1920 × 1080 像素的高清视频，试估算在计算 DCT 时每秒的操作次数。

3. 解释为什么图 7.14 所示的相机 JPEG 系统能被改进为支持 10M 像素图像的相机 JPEG 系统。

4. 假设 FPGA 以 90nm 工艺生产的，试估计表 7.6 所示的 FPGA 和 ASIC 设计的大小（单位为 rbe）。

5. 假设 FPGA 以 90nm 工艺生产，比较表 7.6 所示的 FPGA 和 ASIC 设计的优点和缺点。若 FPGA 和 ASIC 的工艺变为 45nm，请再次比较表 7.6 所示的 FPGA 和 ASIC 设计的优缺点。

6. 假设一个三维图像应用需要处理 k 个未裁剪的三角形，每个三角形的像素个数平均为 p，被其他三角形覆盖的系数为 α。采用环境漫射光照明模型和高洛德着色，分辨率为 $m \times n$，帧率为 f（单位为 f/s）。

（a）计算几何操作中的浮点运算次数。

（b）计算像素值的整数运算次数。

（c）计算光栅化的访存次数。

7. 表 7.10 给出了 ARM1136J-S PXP 系统的性能参数。在 16K 指令缓存和 16K 数据缓存的情况下，数据通路的运行频率最高达 350MHz。32K + 32K 和 64K + 64K 实现方式的运行速度受到了它们的缓存实现的限制。

（a）说明增加一级缓存大小对性能的影响。

（b）说明导致了最大的缓存面积却没有最快的速度的原因。

（c）比较最快的设计和第 2 快的设计，说明哪一种具有更好的性价比，并说明为什么。

表 7.10　ARM1136J-S PXP 系统的性能参数（MPEG4 解码的性能）

一级缓存大小	16K + 16K	32K + 32K	64K + 64K	16K + 16K	16K + 16K	16K + 16K	32K + 32K	32K + 32K
二级缓存大小	—	—	—	128K	256K	512K	256K	512K
速度/MHz	350	324	277	350	350	324	324	324
面积	2.3	3.3	6	8.3	12.3	21	13.3	22
运行时间/ms	122.6	96.7	93.8	70.6	60.7	60	63.6	63.2

注：面积值包括相应的二级缓存的面积。

8. 如图 7.24 所示，两个图像 I_1：$H_1 \times W_1$ 和 I_2：$H_2 \times W_2$ 进行卷积算法，图像的可选的掩模分别为相对同样大小的 M_1 和 M_2，$H_2 > H_1$ 且 $W_2 > W_1$，f_3、f_{12}、f_{11}和f_{22}为纯函数，即函数的结果只与它们的参数相关，与内部状态无关。

（a）在如下情况下计算 M_1，M_2，f_3，f_{12}，f_{11}和f_{22}的值：（1）SAD 相关；（2）归一化相关；（3）高斯模糊。

（b）输出的结果图像 I_c 的分辨率是多少?

（c）产生结果图像 I_c 需要多少拍?

for $y = 0$ to $H_2 - H_1$ **do**

　for $x = 0$ to $W_2 - W_1$ **do**

$$Ic_{x,y} = f_3\Big(\sum_{i=0}^{H_1-1}\sum_{j=0}^{W_{1-1}} f_{12}(M1_{i,j},\ I1_{i,j},\ M2_{y+i,x+j},\ I2_{y+i,x+j}),\ \sum_{i=0}^{H_1-1}\sum_{j=0}^{W_{1-1}} f_{11}(M1_{i,j},\ I1_{i,j}),\ \sum_{i=0}^{H_1-1}\sum_{j=0}^{W_{1-1}} f_{22}(M1_{y+i,x+j},\ I2_{y+i,x+j})\Big)$$

　end for

end for

图 7.24　输出图像为 I_c 的卷积算法

第 8 章　展望：未来的挑战

8.1　引言

伴随着晶体管密度的快速增长，需要对未来的挑战做个预测。一种理想预测是全自治片上系统（Autonomous SoC，ASoC）：它融合了射频识别（radio-frequency identification，RFID）技术与 SoC 技术外加能量转换器、传感控制器和电池，所有这些都集成在一个芯片上。这最主要的挑战在于设计要满足极低的功率（1μW 或更低）和能耗要求。这就需要重新设计时钟、片上存储的组织和处理器核的布置。通过使用薄膜电池、有效的无线电通信、数字信号传感器和微电机系统（Microelectromechanical System，MEMS）可以完成 ASoC 计划。除了满足这样的要求，还需要平衡系统的功耗、RF 和速度。

总而言之，设计时间和成本是 SoC 系统目前主要的瓶颈，并且在未来可能更严重。一种打破这种限制的有效方法是开发一种设计机制，使得组件可以自我优化和自我验证来改善有效性、重用性和正确性——由 ITRS 提出的三大设计挑战。在设计前后的自我优化和自我验证是未来 SoC 设计的核心。

本章共有两部分内容：第一部分描述未来的系统——ASoC；第二部分描述未来的设计过程——自我优化和自我验证。本章所描述的很多挑战的，如果将来遇到的话也是机会。我们在文章中重点标识出具体的挑战。

8.2　未来的系统：全自治片上系统

8.2.1　概述

SoC 技术是微处理器市场的一个延伸，每年增长 20%，未来的增长率将会更高[134]。

典型的 SoC 包括多个异构处理器和控制器，以及多种存储器（ROM、缓存和 eDRAM）。处理器核面向一种或多种特定的多媒体处理。典型的应用包括手机、数字照相机、MP3 和多种游戏设备。

另外一个快速增长的市场是自治芯片（Autonomous Chip，AC）。它们具有很小的处理功耗和存储能力，但是需要 RF 通信和一些自带的电源或能源管理。更

复杂的 AC 可能还包括一些传感器。简单 AC 的有 RFID[205]、智能卡和片上信用卡。

最简单的 AC 是被动驱动的 RFID。它只反映源 RF 的携带者并且通过携带者的电源来驱动显示携带者的 ID。比较复杂的例子如病人监测报警器[31]和 20 世纪 90 年代的灰尘智能探测系统[63,181]。它们都是用电池驱动的 RF 来广播 ID，ID 通过检波传感器输入。

很多智能卡和包括美国的 VISA、我国香港的 Octopus Card 在内的银行卡，除了需要通信的卡以外，都使用 RFID。最简单的卡没有卡上的可写存储，记录只能被集中更新，常基于 Jave Card[234]来启动。基于一些特殊考虑，有一系列遥控识别卡（即 RFID）的标准：

- ISO 10536 紧耦合卡 [close coupling cards (0 ~ 1cm)]
- ISO 14443 近耦合卡 [proximity coupling cards (0 ~ 10cm)]
- ISO 15693 疏耦合卡 [vicinity coupling (0 ~ 1m)]

未来全自动的 SoC 或者 ASoC 是 SoC 和 AC 技术的融合（见表 8.1）。理论上简单，但是工程技术却很困难，因为需要重新考虑处理器的整体架构和实现来优化设计，以便满足低功耗的要求——亚微瓦级。

表 8.1　ASoC 实例

系　统	被动 ID	活动 ID	RF 传感器	ASoC
例子	RFID，简单智能卡	智能卡，活动 RFID	智能微尘；RFID + 传感器	
电源	无	短期电池	电池	集成电池
最大存储量	ROM ID（1KB）	R/W ID + 参数（2KB）		R/W 扩展（100MB）
RF 尺寸	无源的：厘米级	主动 1 ~ 10	10 ~ 20	10 以上
计算	无	FSM	FSM	1 个或多个 CPU

研究 ASoC 的动力来源于智能微尘计划（Smart Dust Project）[63]，在 20 世纪 90 年代启动，对传感器和 RF 领域开展了具有重要意义的工作。此项目把具有 RF 的传感器植入约为 1mm^3 的微尘中。它依靠 AA 级电池作电源。此项目的目标在于检测一种“事件”，如移动目标或者是一个热信号。

ASoC 在计算能力和存储容量方面做了新的扩展，同时在片上整合了一个电源。

根据智能级别，AC 的一个简单分类如下：

1. 简单的在片上用 RF 来标识自己（如 RFID）。
2. 对传感器检波“事件”利用 RF 进行标识 [如智能微尘（Smart Dust）和

智能卡（Smart Cards）]。

3. 利用 RF 对结果的反馈来进行“事件”的检测和处理(分类、识别、分析)。

ASoC 的数据处理能力在降低传感器需要传输的数据量方面很有价值。它使得交互式计算成为可能（如行星探测）；可以用来在偏远地区对稀有鸟类和其他物种进行识别；或者是作为“药丸”用来对肠胃疾病进行诊断。当然，ASoC 尺寸并不是所有的应用场景都很重要。稀有物种的“监听”设备可能不关注尺寸大小，而且要求有足够的电源支持。那么把 ASoC 看成是一个工具箱，为新系统的设计提供可配置的能力来适应千变万化的环境和计算需求。

在接下来将介绍硅技术的演变、电池和能源方面的限制、实现架构、通信、传感器和应用。

8.2.2 技术

就像前面介绍的，接下来的几年里晶体管和存储密度将会增长 10 倍[134]，每平方厘米的晶体管数量将达到数十亿。由于合理功耗的处理器可以用 100 000 级的晶体管数量来实现，对于 ASoC 应用来说有很多可能性。

但是这样的密度是有代价的。太小的设备在传统的工作站的实现中会引起严重的性能问题。通常情况下，掺杂原子的差异（为了生产设备所需要掺杂的原子数量）会导致设备之间的延迟。带有强电场的小结构会导致可靠性问题：导体和绝缘体之间的电迁移。在采用低功耗和低速 ASoC 的情况下，上述问题并不显著。对于可使用的 ASoC 来说，最主要的问题是电池的容量或所能存储的能量。为了处理这个问题，需要回顾一下本书第 2 章所讨论的两个关系，相关的硅面积 A、算法执行时间 T 和能量消耗 P（公式中 k 是常量），有

$$AT^2 = k \quad (8.1)$$

上式[247]反映了硅片面积（晶体管数量）和完成具体操作的执行时间之间的关系。面积越大（晶体管数量越多）速度越快（执行时间越短）。本书第 2 章介绍了执行时间和功耗之间的关系[99]：

$$T^3P = k \quad (8.2)$$

由此可见，随着电压的降低，功耗成二次方下降但是速度成直线下降。但是对于晶体管来说，延迟和电压之间是非线性的关系。就像本书第 2 章介绍的，是三次方曲线：

$$P_2/P_1 = (F_2/F_1)^3 \quad (8.3)$$

所以，如果把主频提升一倍，那么功耗将提高到 8 倍。当评估规划微瓦级芯片的频率时，式（8.2）的适应性就没那么精确了。当今最好的功耗设计是用 1W 的开销达到 1GHz 的效果（相当于达到 1000MIPS）；这也许是乐观的情况。以 10^6 量级来降低功耗就得以 100MHz 或 10MHz 的量级来降低频率。在过去的 2 年里，

传感器处理器的设计几乎已经达到了 0.5MIPS/μW[271]。但是这距离 10MHz/μW 的目标还是非常的大，硅缩放技术也许能弥补这个差距。

挑战

$T^3P = k$ 这一定理在微瓦级还适用吗？

这一定理在普通计算条件下是正确的，但是怎么把它扩展到微瓦级呢？需要什么样的电路和策略呢？

8.2.3 功耗

设计稳定高效 ASoC 的关键在于解决功耗和使用寿命的问题，这两个问题都和电源（电池）有关。电池可以是一次性的也可以是重复充电的。对于 ASoC，蓄电池要从周围环境中吸收能量。电池的能力以蓄电小时来衡量，转换成 1.5V 下的 J（W/S）。电池的蓄电能力和可充电性都取决于体积，包括表面积和 ASoC 每平方厘米的重量。

表 8.2 给出了 ASoC 用电池对比，是三种常用的电池类型：印制电池（printed battery）[47,203] 和薄膜电池（thin film）[67] 可以直接整合到 ASoC 中（一般在背面）；钮扣电池一般是外设，直径在 1cm 以内。印制电池是通过在平滑的表面上喷印一些特殊的材料形成的。薄膜电池是和 ASoC 一起集成到硅片上的。

表 8.2　ASoC 用电池对比

类　　型	单位能量/J	可否充电	厚度/μm
印制电池	$2/cm^2$	否	20
薄膜电池	$10/cm^2$	可	100
钮扣电池	200	可	500

表 8.3　能量转换源[173,195,206]

源	转　换　率	备　　注
阳光	$65mW/cm^2$	
环境光线	$2mW/cm^2$	
张力和声音	晶体结构的形变产生电压	电压效果
RF	10V/m 的电场产生 $16\mu W/cm^2$ 的天线	本书参考文献[266]
温差（珀尔帖效应）	40μW 每 5℃温差	需要有温差

挑战

以 $1cm^2 \times 100\mu m$ 的体积能提供 100J 以上能量的电池技术才可以集成到硅片上。

很多应用都急需微电池技术。

能量的来源有很多途径（部分见表 8.3）。一般情况下能量源的体积越大电量就越大，在很大程度上还取决于系统的环境是否适宜吸收能量。

假设 ASoC 消耗 1μW 的能量（工作情况下），所需要的充电周期有所变化（见图 8.1）；占空比在影响 ASoC 的服务能力方面起到了很重要的作用，前提是被动传感器能侦测到信号然后再启动系统进行分析。

挑战 利用现有的可用技术，从更多的能量源转换电能。

到目前为止，对于能量转换（见图 8.2），大家的关注点还局限于光和 RF 技术（如 RFID）。那么就需要对取舍进行全面的研究，特别是在微瓦级转换时。过去，认为这种低能耗的转换没有使用价值，但是对于 ASoC 来说这是很有意义的。

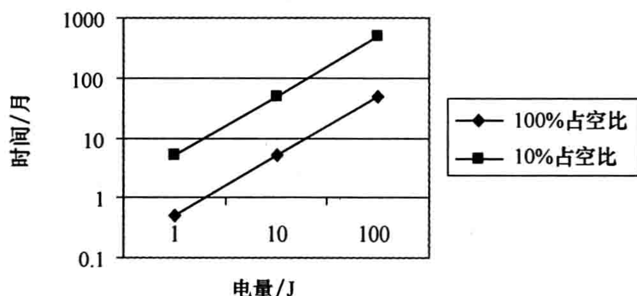


图 8.1 对于 1μW 的不停消耗功率最大的充电时间间隔

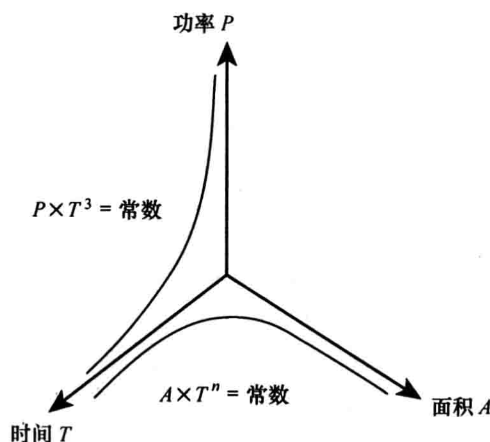


图 8.2 面积、时间和功耗之间的关系

8.2.4 全自治片上系统的外形

ASoC 逻辑结构如图 8.3 所示，包括电源、传感器（一个或多个）、主计算

机、存储单元及通信模块。ASoC 与之前 RFID 传感器的不同之处就在于有计算能力和存储单元。正由于这些功能使得 ASoC 具有分析和识别能力，在与外界交互之前可以做出综合的反应。

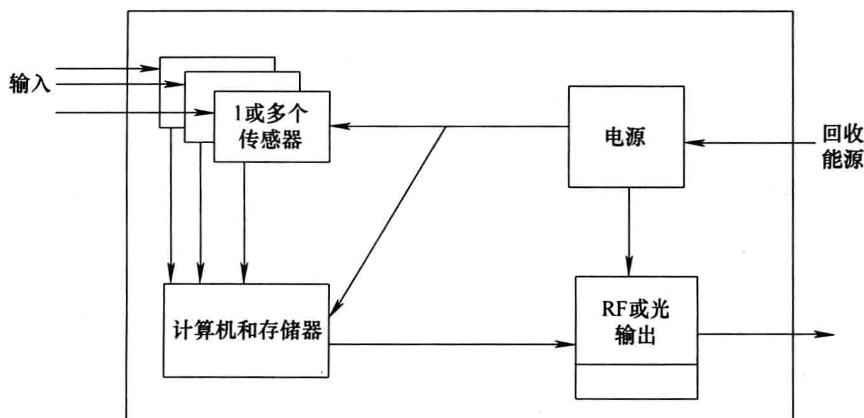


图 8.3　ASoC 逻辑结构

在物理上，ASoC 只是一个硅片，表面积在 1cm^2 左右。表面积的大小受限于成本，而成本又由缺陷密度决定。现有的技术对于 1cm^2 或稍小一些面积的硅片，可以保证非常好的优良率。对于 1cm^2 以下硅片，成本主要集中于测试和处理，所以对于大多数应用来说首选还是 1cm^2。因受限于晶片的制造条件，硅片的厚度一般是 600μm。在背面增加一个薄膜电池会增加 50μm 的厚度。最终的 ASoC 一般在 65mm^3，重 0.2g。由图 8.4 可知，它大概有 10 亿的晶体管。这些晶体管构成了传感器、计算单元、存储单元和 RF，电池在背面。

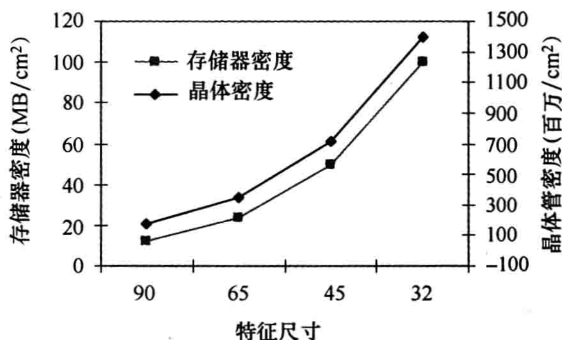


图 8.4　ITRS[134] 晶体管密度预测

8.2.5　计算机模型和存储

对于功率只有 1μW 的计算机，其微体系结构与传统的处理器有以下很大的区别：

1. 异步时钟。最少化数据状态的转换来降低动态功耗。在整个时钟系统中，可能只有 1/10 的异步转换需求。

2. VLIW 的使用。由于晶体管数量不限而功耗受限，所以要尽量利用所有可用的并行度来提高性能。

3. Beckett 和 Goldstein[39]已经证明通过小心的设计（减少当前的活跃驱动和管理的漏损率），有可能把整个硅片的功耗减少到一个区域的功耗。这会牺牲最大的工作时钟频率，但是额外的面积可以通过并行化体系结构来抵消掉。

4. 最小的简单缓存系统。如果处理器每 0.1μs 处理一次，而存储单元的访问延迟在 1~10μs 之间，那么存储单元和处理器具有比较接近的时间周期。小型的指令缓存和数据管理缓冲器更适合于特定应用。

存储单元是系统不可缺少的组成部分，因为即使没电了还是会记录数据。当前的存储密度可以到达非常卓越的访问速度——10μs，而且 ASoC 的存储能力在 16~64MB 之间。

基于当前的配置技术，闪存和 CMOS 技术非常不兼容，并且受限于片外技术的实现。但是，现在有很多闪存的变体，专门为了兼容通用的 SoC 技术。SONOS[233]是可永久存储的例子，Z-RAM[91]是代替 DRAM 的例子。这两者都不受传统闪存重写时钟的限制（大约 100 000 次写）。

即使闪存不存储时没有功耗也不会被使用，因为当它被访问时能量消耗随着活跃数据存储空间的大小按正比上升。更确切地说，是与每一位和字单元相连接的存储单元数量（假设是二维的结构）按正比。在 ASoC 里面，这意味着存储单元可以分成更小的部分，使得功耗和访问时间都达到最有效。

8.2.6 RF 和激光通信

ASoC 的最大的挑战之一就是通信。比较常用的两种途径有激光和 RF 通信。

挑战

极低功耗的处理器在微瓦的功率大概能达到传统处理 1/100 的性能。

这就需要从晶体管级（最小化静态功耗）到新的电路技术（阈值以下或者隔热的电路）对处理器设计重新进行设计、新的时钟和全新的体系架构。

8.2.6.1 激光

激光和硅的融合是一门新兴的技术。最近的一项研究[85]利用硅波导管把激光器直接连接到硅片上。利用激光器实现光通信具有可能性的同时也有很多困难。

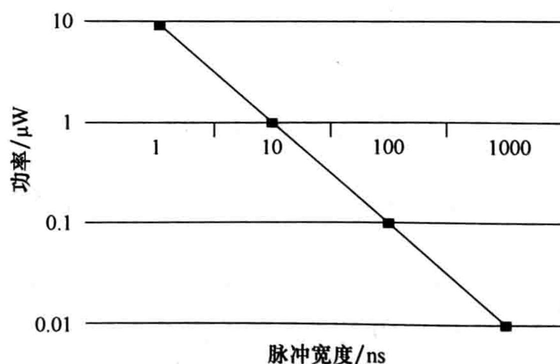


图 8.5 图片信号敏感度与脉冲宽度的关系

光传感器非常灵敏[136]；1μW 的功率能达到 100MHz（见图 8.5）。困难在于信息接收受到周围环境光线（噪声）的影响。一般情况下信号的强度要高于周围干扰的 10 倍以

上。另外一个难点是光束的发散（特别是在激光二极管中）。这需要利用光学原理来校准发散的光束[192]。

光束不能太窄（聚焦），因为通信接收端必须在时间和空间上完全同步。对于相干的窄光束，光必须散射以容忍源和接收端之间传播的振荡光振荡会在 d 距离内导致垂直或水平方向的 α 角移位。这就导致接收端 δ 的不确定性。所以，接收端必须具有一个 $R\times R$ 的信号接收区域，如图 8.6 所示。

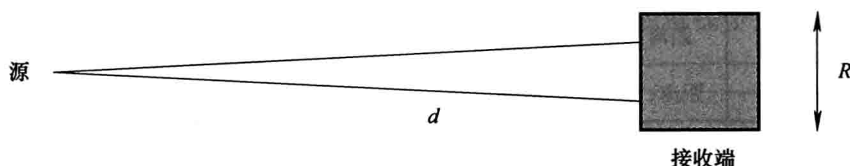


图 8.6　光通信的信号接收区域与距离

因为在 x 的负轴和 y 的负轴的情况下，$R>\delta=\alpha d$，信号的丢失率为 k（$1/d^2$）。

考虑到以上限制，利用激光自由空间（而不是光纤）进行通信，对于 ASoC 来说，很可能会成为第二值得研究的领域。

8.2.6.2　RF

智能微尘计划的研究工作几乎是这个方面最有意义和最有用的[63,181]。它们把低功耗的 RF 集成到一个 SoC 中。它们的研究成果可以总结如下：

1. 此可行性研究实现了一个无线电收发机，可以在 20m 的距离 25nJ/bit 的功耗下，达到 100Kb/s 的速率。相当于 10^{11}bit/（J/m）。1J 的电池能量可以使得 1Gbit 的数据传输 1m[181]。

2. 以每米为基础来表示所传输的数据量（如上所述），似乎表明了在信号损失和传输数据之间有一个线性关系。其实这是不对的。对于光来说，RF 信号的强度至少是传输距离 d 的二次方的函数；而且它还是频率 f 的函数。RF 信号顶多与 $k[1/(fd^2)]$ 成比例。在很多情况下，信号会被反射并以多种不协调的模式到达接收端。这种多路径的信号意味着额外的信号损失。通常被表示为 $k[1/(fd^2)](d_0/d)^n$。其中，d_0 是标准距离（通常为 1m），n 通常为 3 或 4。

3. 以低于 1mW 的功率进行通信不但可行而且可能会商业化。对于典型的低于 1% 的占空比，平均消耗 1～10μW 的能量。

4. 另外，还有大数据包开销（包括启动和同步、启动信号、地址、包长度、加密和纠错），短消息只有 3% 的有效载荷，所以生成较少数量的较长数据包会更好。

5. 根据 2 和 3 的分析，系统设计者会尽可能地减少信号传输的次数，并尽可能地增加信号数据包的有效载荷。

8.2.6.3 激光和 RF 的通信能力

表 8.4 给出了通信技术对比，总结并对比了激光和 RF 每焦耳的通信能力(数据量或者数据位数)。虽然激光通信每焦耳可传递更多的数据，但是它本身的局限性限制了它的使用。

表 8.4 通信技术对比

源	损耗	10m 的通信能力/[bit/(J/m)]	备注
激光	距离；外界光干扰	$10^{10}\sim10^{12}$	
1GHz 的 RF	距离；多通道，频率	$10^{8}\sim10^{11}$	[63, 181]

挑战

可适配的最优化通信包括 RF 的超定向天线和激光的可自动适配的特殊同步机制及传送。

协议需要支持定向短波的初始化传送，以便激活接收端和发送端，来调整到同一直线上达到最优的传送路径。

8.2.6.4 ASoC 网络

在很多情况下，多个 ASoC 组成一个网络，中间经过多个节点把接收端和发送端连接到一起。对于这样的系统，为了实现可连接，节点之间的距离有一个最大的上限。这个最大上限与路径损失因素有关。对于 RF，智能微尘做的实验证明，随着 n 从 2 增加到 4，这个最大的上限距离会从 1km 缩短为 10m。还有很重要的一点，随着数据位在网络中的传输会有同步开销（有时会很大)。时间和空间的同步需要适配和发送的开销。仅对于时间同步，每条信息大约需要 100bit。理想情况下，系统应该具有数量少但比较长的信息来最小化开销。

挑战

通信技术需要最小化同步开销（包括时间和空间)。

为了利用短消息进行有效的通信，至少要把同步开销减少到大约 10s 的位传输。

8.2.7 传感

8.2.7.1 视频

视频和动态传感器一般由光敏二极管阵列组成，阵列大小从 64×64 到 4000×4000 甚至更大[144]。每一个二极管代表图片中的一个像素（针对黑白电子图片)。对于彩色和多谱图片，需要三个甚至更多个二极管来代表一个像素。为了节省功耗和减少状态的转换，ASoC 实现视频处理器时对于阵列中的每一个像素可能只

用一个二极管单元。

对于其他的图片识别和运动侦查识别，非常有必要找到与参考图片一致的地方或者图片中的某一帧块相对于前一张图片的运动方向[65]。在两张图片中适配出相同的场景或轮廓需要把图片分割成不同的帧块。然后逐块地与参考图片或前一张图片进行螺旋式的比较。每一个比较包括计算绝对差之和（Sum of Absolute Difference，SAD）指数。当具有最小 SAD 指数的图片被找到，识别就结束。但是图片的识别应该在毫秒级，对于视频传感器来说，挑战就在于要满足相关快速移动目标的计算需求（见图 8.7）。

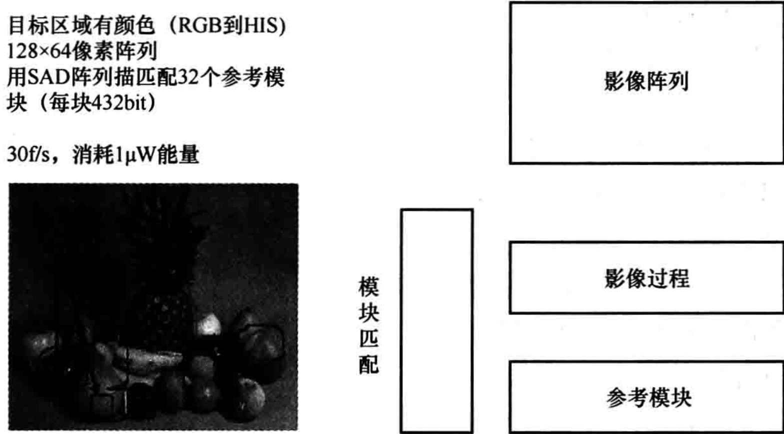


图 8.7 视觉处理

很显然，图片传感器可以集成到 ASoC 中，利用光学把不同的对象集中成不同的光束可以改善性能。

8.2.7.2 音频

如上所述，硅晶体的压电效应，可以用来记录声音，这是很多声音识别系统和扩音系统的基础。另外，对于某些特殊的应用，如助听器，有时候更重要的是模拟耳朵的功能。很多耳蜗片上系统已经被实现，通过使用一串低通的滤波器仿真耳蜗的功能。在一个硅晶体的实现中[251]，360 个由双向低通滤波器组成的单元排成一个阵列。当语音识别需要的时候，耳蜗模式会被选择：它们组成听觉系统的前端信号处理部分，分割出声波并转换成耳朵能识别的频率范围（见图 8.8）。

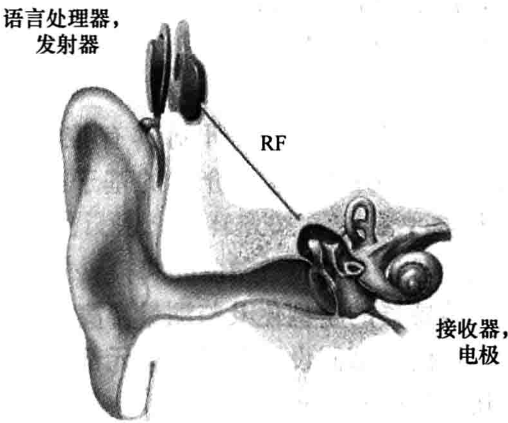


图 8.8 音频处理（图片来自维基百科）

由于可听见的声音频率相对比较

低，所以对于 ASoC 几乎没有实时的限制。

8.2.8 动力、飞行及果蝇

当然，最终的 ASoC 不但可以移动还可以飞行。每次仅 0.2g 的载重需求，对于具有 MEMS 的 ASoC 来说应该问题不大。利用 MEMS 和纳米发动机可以在表面上固定和移动 ASoC。在表面上移动需要的能量，只是启动（加速）和克服摩擦力所需的能量。1J 的能量转化成 10^7erg⊖的功。1erg 是移动 1cm 重 1g 的物体所需要的能量。所以，较低频率（低于 1% 的占空比）的慢运动（大约 1～2cm/s），不会导致重大的能量消耗。

飞行的动力系统是目前最复杂的。已经有很多针对小型自动飞行器的动力研究[273]。飞行对于 ASoC 来说有许多挑战：功耗、视觉识别（避开障碍物）、环境（风等）及通信。虽然可飞的 ASoC 还很遥远，但是这种系统和任何小果蝇一样具有相同的可行性[209]。

非常有趣的是，这里介绍的高要求的 ASoC，与像果蝇这样的生物来比（见图 8.9），还是显得非常保守的。果蝇一般长 2.5mm，具有 $2mm^3$的体积，体重不超过 20mg，通常只有 1 个月的生命周期。



图 8.9 果蝇（图片来自维基百科）

但是果蝇的视觉处理能力非常惊人。它有 800 个视觉接收单元，每个单元具有 8 个紫外线的彩色感光器（每个感光器使用 200 000 神经元，总共约有 100 万个神经元）。据估计，它的视觉系统功能比人类的强 10 倍。再加上嗅觉、听觉、学习/记忆，以及与其他节点（果蝇）的通信，它简直是一个完美的 ASoC。它的飞行能力可以进一步描述成：翅膀每秒可振动 220 次，每秒可飞行 10cm，可以在 50ms 内旋转 90°。它的能量来自腐烂的植物果实。

现在已经有关于研制具有控制系统的机器苍蝇来模拟真实苍蝇的提议[255]。很明显，这里介绍的基于硅的 ASoC 设计者还有很多地方需要从果蝇身上学习。

挑战

传感器的小型化及与测量温度、拉力、移动和压力的传感器一体化。

目前，都假设这些基本单元都在片外，而且比较大。问题在于如何把它们小型化并集成到 ASoC 中。

⊖ erg：尔格，$1erg = 10^{-7}J$。

8.3　未来的设计流程：自我优化和自我验证

8.3.1　动机

本章剩下的部分介绍改进包括 ASoC 在内的高级 SoC 的设计方法。

好的设计是非常有效的，而且能满足需求的。通过优化提高效率，同时要通过验证来证明需求被满足。不幸的是，现有的很多设计要么没有效率，要么不正确，甚至是既没有效率也不正确。

优化和验证在设计的所有抽象级别中都被认为是非常重要的。最近 ITRS 把“成本驱动的设计优化”和“验证和测试”列为设计中三大挑战之中的两个，剩下的一个是“复用”。

未来如何满足这三大挑战呢？假如在设计中用到的结构模块具有自我优化和验证的能力。那么一个新的设计可以通过以下步骤完成：

1. 抽取出设计所需要的具体属性，来定义具体的需求，如功能、准确度、功耗及倾向的具体技术。

2. 开发或选择一种能满足需求的架构，并研究针对具体结构模块的实例。

3. 首先确定现有的模块是否能满足需求；如果不能，要么开始一个新的研究，要么开发可优化的并且可验证的结构模块，或者是把需求调整成可以实现的功能。

4. 在确认经过优化和验证后的设计能满足需求以后，规划优化和验证的步骤，来确保设计能实现自我优化和自我验证的功能。

5. 一般化具体的设计和相应的自我优化和自我验证的能力，来增加它的适用性和可复用性。

核心的理念就是在设计的过程中保证可以自我优化和自我验证：从具有这种特性的组件开始，最后综合成的设计也具有自我优化和自我验证的特性。下面将会更多地介绍这种方法的具体细节。

8.3.2　概述

通过优化可以把显著（obvious）但无效的设计改成有效但不再显著的设计。然后通过验证可以证明，如优化保留了满足前提条件的功能行为。在设计中常犯的一个错误是在优化过程中忽略了前提条件。验证还可以用来检验设计是否具有满足相关标准的具体特性，如安全性和保密性。

当优化和验证与通用设计结合时，一般通过三种主要途径来支持复用。第一，经过优化之后的通用设计，提供具体的细节介绍，使得设计者能关注现有的优化选项及他们的效果。第二，通用设计提供多个层面的设计说明，从算法、结

构到具体的技术细节。第三，验证过的优化流程，增强了原设计的正确性。在设计被复用之前，必须先保证正确性。一旦出现错误，就可以确定到底是验证本身的错误，还是使用设计的环境超出了验证的有效范围。

要以更广义的角度来看待自我优化和自我验证。一种方法是规划一个设计（可以包括硬件和软件）和具体实现应该具有的核心属性。这些属性包括功能的正确性、类型的通用性、计算上溢和下溢异常的避免等。描述可以包括借助特殊工具在本地或远程对设计进行优化和验证的具体方法。很多方法都可以借助有效的上下文信息而应用到自我优化和自我验证中，如脚本驱动的技巧和机器学习的步骤。设计者可以致力于优化和验证特殊的部分，如想通过最小的设计来计算AES，用512位的密钥在500MHz的频率下加密128位的数据流。

布局前后的上下文如表8.5所示，推荐的设计流程在发布前后都包含自我优化和自我验证。发布前，编辑生成一个初始化的实现和它的特征描述。特征描述包括设计已有的优化和验证是如何进行的，还包括进一步优化和验证的因素；这些优化的因素可以在系统调度之后运行时进行，来改善有效性和正确性。

表8.5 布局前后的上下文

	布局前	布局后
焦点上下文	设计工具环境，静态	设计效率的操作环境，动态
获取上下文	来自于影响工具环境的参数	来自于输入数据，如传感器
优化、验证	优化、验证布局前的静态设计	根据条件优化
规划	规划布局前的优化、验证	规划如何满足布局后的目标
外部控制	频繁	不频繁

自我优化要基于上下文进行。在发布之前，上下文是设计工具的环境；通过制定可以改变设计工具的参数来设置上下文。另外自动化能力，自我改善的同时，还要努力计算出如何组合组件和工具得到最能满足需求的设计，设计者可以随意地控制工具来确保这种自我优化和自我验证在正确的方向上进行。相比之下，在发布之后这种额外的控制就会少很多，如当设计是飞行器的一部分时。总之，发布前的任务非常具有战略意义，要能提前预测出在运行过程中可能发生的事故；发布之后的任务主要在于战术，必须要在可行的行为集中选择合适的操作来应对不断变化的运行时的上下文。

这个方法主要有三个好处。第一，通过自动化验证过程，可以提高对正确性和可靠性的信心。第二，通过自动化优化过程和扩展运行时系统的自适应性可以改善设计的有效性。第三，通过使能设计的复用、优化和验证，可以提升生产率。

但是，采用系统的设计复用，特别是当包含自我优化和自我验证时，比只做一次性的设计需要更多的初始化工作。设计者需要妥善组织、生成并记录设计的

细节，直到复用值得花时间做为止（见图 8.10）。此外，在发布之后需要很大的开销来支持优化和验证。但是，从长远来看，在可复用和适用性设计方面的努力都会在设计的有效性和生产率上获得很大的收益。

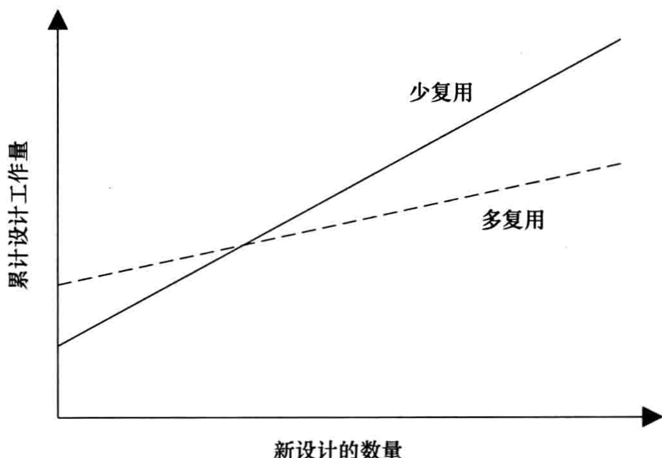


图 8.10　设计工作：复用的效果

8.3.3　部署前

在部署之前，设计者需要有预期设计的特征描述，并且有构建这些建模块和它们特性的途径。任务是开发一种架构来定义被选定的模块如何被实例化并组合生成一个初始设计，不但要满足需求，同时还能在部署后的运行过程中被进一步地优化。必须要仔细认真地规划部署后的优化和验证，来避免不可挽回的代价。

假设在部署前编译时，有以下情况：

1. 现有的计算资源足够支持设计和工具；

2. 但是，可做的优化和验证却有限。例如，一些对优化有用的数据只有在运行时才知道，而且如果计算出这些数据所有可能的值也不现实。

挑战

获取设计和上下文的组合描述在不同层次的抽象，包括优化和验证的特性。

组合是一种非常方便的复用方法，但是它并不是最直接的方法，不同于采用像数据流这样的基本的通信方法。尤其是在组合异构组件之前，可能需要转换来支持通用通信和同步基础设施。系统级的设计创作非常具有挑战性，因为不但设计本身是组合的，而且对应的优化和验证也同样是组合的。

举一个简单的例子，假设 n 位加法器的两个操作数中的一个是常数，它的值在部署之后运行时才知道，想利用常数传播来优化这个加法器。但是，要提前计算出所有 2^n 个可能的配置是很不现实的，除非 n 的数值比较小。幸运的是，如

果针对位片架构，那么就足够提前计算出 n 位中每位仅有的 2 个配置，在运行时如果已经知道相应的值，那么合适的配置就可以在正确的时间放置到正确的位置上[216]。

设计者可能需要优先排列或修改相应的需求直到找到可行的实现。例如，当需要最优的有效功耗设计来满足特殊的定时约束或者用最小的设计来满足数字精度。其他的一些因素，如安全性和保密性，可能同样需要考虑到。

假设部署前优化是为了在将来部署之后的适当情况下进一步进行优化，以下是一些在部署之前可以做的优化的例子：

1. 选择一个可以实现设计的电路技术。比较通用的两种技术是 ASIC 和 FPGA。技术的选择在于设计量和灵活性（见图 8.11）。例如，基于（Cell）的 ASIC 更倾向于减少设计量，因为它们有大量不可重现的设计成本。而 FPGA 是另一种围绕结构性的 ASIC 的方法。ASIC 技术可以用来扩展常用指令[29]和可配置的缓存[76]来实现自适应指令的处理器，所有的可配置选项必须在制造之前是可知的。自适应指令的处理器也可以基于 FPGA 技术实现[77,269]，可以使它们以速度和面积作为开销来支持可配置的同时具有更多的灵活性。

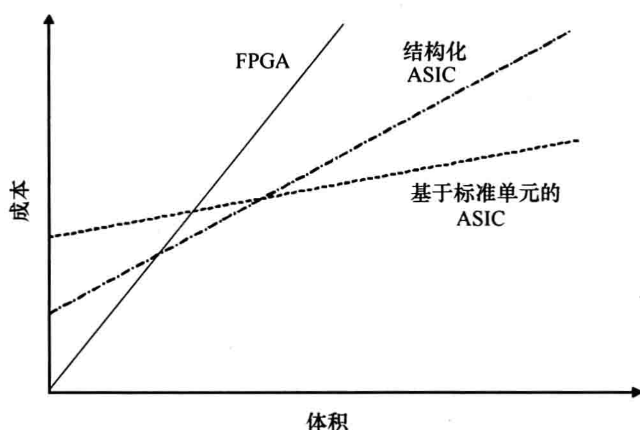


图 8.11 FPGA 和 ASIC 技术的成本和体积对比

2. 为可配置单元选择粒度和同步规则。当前的商用 FPGA 都是具有一个或多个全局时钟的细粒度设备。但是也出现了一些其他的结构，如包含多位 ALU 组成的可以并行执行阵列的粗粒度设备[25,80]和基于同步技术来提高扩展性的结构[52]。一般情况下，细粒度设备具有更好地通过定制来很好地满足需求的机会。例如，如果需要一个 9 位的 ALU，FPGA 中 9 个位级的单元可以通过配置来组成 9 位的 ALU。对于包含 8 位 ALU 的粗粒度设备，则需要两个这样的单元。但是，细粒度设备在速度、面积和功耗等方面具有更大的开销，因为细粒度设备具有更多可配置的资源。相反，粗粒度设备由于可扩展性差，开销也小。

3. 对于支持传统指令的指令处理器[29,77]，通过选择传统指令的粒度可在速度和面积之间达到比较好平衡。粗粒度的传统指令比细粒度的速度快但是需要更多的面积。例如，如果使用（a）一条粗粒度的传统指令或者（b）50 条细粒度的传统指令，可达到相同的效果。那么，（a）需要更少的指令预取/译码操作，

性能一般比较快，而且有更多机会来定制传统指令来实现具体的需求。但是，由于指令粒度越大，专用性越强，粗粒度传统指令的复用机会也就越少。

4. 通过改变处理单元的数量、流水线的级别及每个处理单元之间的任务共享度来改变并行度和软硬件的分区，进而来匹配性能或体积的限制。同样这里也要考虑到如速度、控制逻辑的大小及片上存储等多种因素，以及连接存储、传感器等其他单元的接口。下面举一个例子，针对归纳逻辑编程应用而定制的多核架构 FPGA 的加速比随着处理器核数量的变化而变化（见图 8.12）。由于 FPGA 的片上存储是固定的，增加处理器核的数量就会相应减少每个核的缓存存储；所有线性加速只能到 16 个核。在这个最优点之后，再增加核的数量就会降低加速比，因为每个核的缓存已经变得太少。

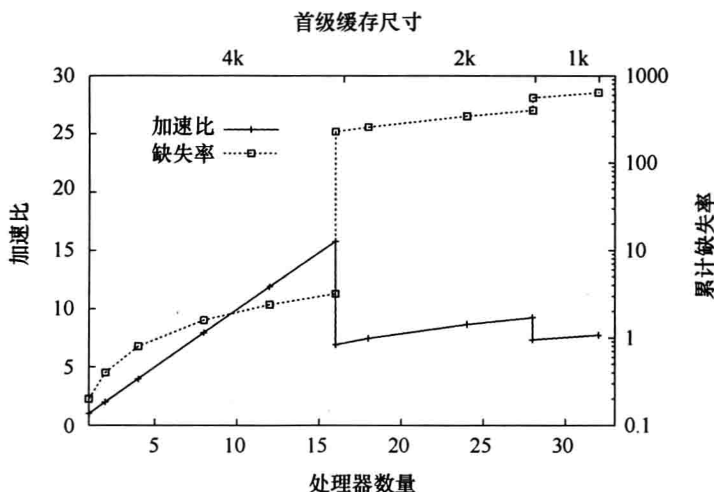


图 8.12　基于 Arvand 多核处理器系统的 XC2V6000 FPGA 上加速比和累计缺失率随着处理器核数量的变化

5. 选择数据的表示形式和对应的操作。在多种算术表示方式中采用平衡设计技术非常流行。例如，冗余算法更倾向于取得更快的设计，因为不需要增加面积来取得进位链。由于细粒度的 FPGA 支持任何字长度的设计，多种静态和动态的字长优化算法，可以支持在性能、面积、功耗和如信号-噪声比准确度等方面取得最好平衡的设计[62]。同时也要考虑支持算术上溢和下溢异常的模型和工具[153]。

6. 在物理器件上为处理和存储单元选择布局策略，如经常需要交互的组件紧挨着放置，来改善性能、面积和功耗。可以通过启发式合并和基于搜索的自动调谐器[27]来生成和评估不同的实现方式实现自动优化布局；这种方法需要考虑多种结构上的约束，如存在嵌入式计算或存储单元[36]。

以上的每一个例子都有通过验证来获得收益的方面，从高级编译[43]到扁平化程序（flattening procedure）[168]和布局策略[196]。现有的验证平台[236]使得应用的验

证工具一致化，如符号化仿真器、模型检查器和定理证明器。这些平台表面支持复杂设计自我验证的可能性，但是还需要做很多工作来证明设计能利用各种技术和能通过多层次的抽象。另外，这些平台和工具很多可以通过自动调谐来获益[121]。

部署前一个重要的任务是规划部署之后的自我优化和自我验证。这个计划要基于部署之后具体有多少有效的运行时信息来制定。例如，如果某个设计的输入是不变的，这些不变的输入可以由设计通过布尔优化和重定时传播。这些技术可以被扩展应用到覆盖定位策略，来产生紧凑布局[168]的参数描述。另外一种可能，是选择适当的结构模板来促进运行时资源的整合[211]。

部署之前，如果验证已经覆盖了优化和所有其他的部署后的操作，那么部署之后就没有必要进行进一步的验证。但是，如果这些优化和验证虽然有效，但是并不被某一特定的设计支持，那么这些优化和验证可能很少会发生，所以这些被优化和验证过的设计会在合适的时间被安全地下载到运行中的系统，尽量减少中断系统的服务。

挑战

开发各种技术和工具来指定和分析自我优化和自我验证系统的需求，以及自动优化和验证操作的方法和数据的表示法。

相关优化技术包括调度、重定时和字长优化。另外，相关验证技术包括程序分析、模型检查和定理证明。它们有效的优化组合，与探索合适的计算模式及相关影响的新方法，可以使得有效设计事半功倍。

8.3.4 部署后

优化的目的是调整设计，使其能最好地满足需求。但是，越来越多的需求在部署使用之后不再与部署之前一样，如有新的标准需要满足或者新的错误需要处理。所以需要可升级的设计来支持优化以便满足不断增长的需求。除了升级以外，部署之后的优化同样要满足资源共享、错误移除及适应运行时系统的具体要求，如基于噪声变化选择合适的纠错代码。

很明显，任何可编程设备都要在部署之后可以被优化。就像前面描述的，细粒度设备比粗粒度设备具有更多机会来调整自己，当然开销也较大。

接下来重点介绍两种部署后优化的方法：条件适配优化和自主控制优化。这两种情况，任何不可信的部署后优化都应该被轻量级的验证来检测出来；可能用到的技术包括带证明的代码检测器（proof-carrying code checker）[252]。这些检查者支持的参数可以俘获特殊操作的安全状况。有一个证明规则的集合可以被用来建立可接受的方法来证明安全状况。

正如前面提到的，是否需要重负荷优化和验证，对于有些任务是可能需要的，这些任务需要通过独立可信的代理远程执行并通过安全路径下载到可用的设备中，

基于数字签名可能可以识别发送者身份。另外，通过超时设备来证明自我优化和自我验证程序的无休止性是需要非常谨慎的，而且在完成之前也不会产生结果。

除了拥有超时设备，部署之后的验证在处理其他形式的异常方面是非常有效的，如验证失败或者算术错误。应该有错误恢复程序和能决定是否避免或纠正未来类似错误的技术。对于有些应用，片上调试工具[120]非常有用；这些工具可以自己适配来满足不同应用操作和缓存的需求。

8.3.4.1　条件适配优化

一种在不断变化的执行环境中基于部署后优化来获取收益的方法是，不断地适配变化的条件，如温度、噪声和流程变化等。例如，动态重配置维特比译码器的纠错卷积码来适配信道噪声条件的变化，可以降低70%左右的译码功耗开销，并且不降低译码的准确度，这一点已经被证明[241]。

图8.13给出了SPECviewperf benchmark 9工作指标上限的最优配置调度，包含了一个重配置时间表最优适配SPECviewperf[232]测试程序的阶段行为。一个程序阶段的工作集保持恒定的一个间隔。其目的是支持一种动态优化规则，利用程序间隔信息在运行时优化设计。这种规则包括一个硬件编译体系来生成配置增强程序分支的准确性[231]和其他不同执行阶段的优化机会，以及一个运行时系统来管理配置的交互以此来维护阶段转变之间的优化。具体思路是，在一个特定的程序段中，加快经常用到的硬件分支的速度；当检测到下一个阶段的开始时，重新配置硬件来优化这个新的程序阶段。

图8.13　SPECviewperf benchmark 9工作指标上限的最优配置调度
（虚线和实线分别表示内部和外部循环时间的可能拐点）

为了利用程序阶段的行为进一步优化性能，部署后的优化仍然有优化功耗开销的潜力。图8.14给出了功耗开销随着时间的一种可能变化。与静态设计相比，部署后可优化的设计，可以根据特定情况配置成最低功耗开销的设计，即使在设备被重配时有一个功耗峰值。相关技术已经被应用到FPGA中，来动态校正他们

运行时的时钟速度[49]或者是进行动态的电压调节[57]；相关方法已经在微架构中有应用[73]。这些技术还可以从部署之后的运行时条件中获益，就像在深亚微技术中适应处理变化的影响一样。

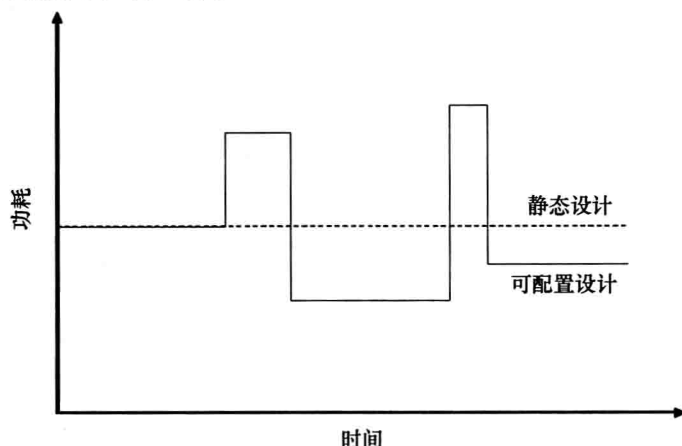


图 8.14　功耗开销随着时间的一种可能变化（两个段时间的高功耗表示在运行优化过程中的两个配置的功耗）

一种支持条件适配优化的有效方法是，把特定领域中的定制整合到一个高性能虚拟机上，这样对虚拟机来说部署后设备的静态和动态信息都有效。这些信息可以在多种条件下应用到自我优化和自我验证中。例如，基于库代码的特定属性和部署后操作的上下文来优化硬件和软件库的使用路径。

8.3.4.2　自主控制优化

“自主计算”[139]已经被应用到系统中用来支持自主管理、自主优化甚至自我修复和自我保护。它受到计算机系统不断上升的复杂性所驱动，需要很努力地来安装、配置、调谐和维护。相比之下，侧重于设计处理来支持并获益于自我优化和自我验证的组件。

一种针对自我优化的改进后的控制策略可以基于事件驱动，在隐藏配置延迟的同时，针对运行时条件来生成软件代码和硬件配置信息以便及时重配置。一个方向是基于组件的元数据（metadata）描述[138]，为具有适应能力的组件去发展理论和进行实践，包括软件单元和硬件单元。这些描述说明了现有的优化，并提供一个性能模型和一个利用组件的元数据，针对给定上下文来找到并配置最适宜实现的元程序。这个工作可以与当前可定制的硬件编译技术合并在一起[245]，这样就可以在基于契约的方法中使用元数据的描述，与研究具有适应能力的软件组件一样。

另外一个方向是研究所需自主行为的高级描述，以及这些描述如何被用来生成活跃的计划。一个活跃的计划通过为每个状态的每个目标分配一个行为来适应

变化的环境，通过这个行为可以实现相应的目标[238]。动态重配置可以被相应的计划来驱动，计划制定了相关配置所应该支持的属性。

对于自主控制优化，其他有希望的方向包括机器学习[6]、归纳逻辑程序设计[89]和自我组织的特征图[200]。实际自我适应系统的例子，如那些针对航天的任务[140]，同样应该被研究来探索它们的潜力，以便扩展使用性和推动理论的发展。为这些优化方法找到一个合适的可证实的概念。

挑战

找到发布前后在协同优化和协同验证之间提供最优分块的策略。

在发布之前完成的工作越多，发布后的设计针对给定的应用就越有效，虽然会损失灵活性。在发布前和发布后的优化和验证之间获取正确平衡的策略非常有用。

8.3.5　规划和挑战

就眼前来说，要理解如何组合自我优化和自我验证组件，以使得最终的混合设计仍然可以自我优化和自我验证。一个关键问题是在相关设计模型和模型表示之间保证理论和实际的结合，如同对应优化和验证步骤之间，在语义模型和不同工具接口的通用性之间保证一致性。

在特定的应用领域首先开始研究自我优化和自我验证设计，看起来是一个不错的想法。从这些研究获取经验，来总结出超越特定应用特性的自我优化和自我验证设计的范围和领域的基本原理和理论。

另外一个方向是研究一种基于平台的方法来发展自我优化和自我验证系统。很多有前途的工作[236]已经应用在合并复杂设计的验证工具中。这些工作提供了一个基础，基于此可以进行自我优化和自我验证的进一步研究。开放的资源库使得设计和工具可在不同设计之间共享，这是非常有用的。特别要指出的是所提出的这个方法不但可以从验证过的软件库资源中收益，同时也对库做贡献[43]，当前作为英国“Grand Challenge”可信系统升级项目的一部分正在被研究。

很显然，要挖掘出自我优化和自我研究的潜力还有很多研究工作要做。要加快自我优化和自我验证在未来的发展，需要在多个领域里取得进步。

挑战

到目前为止，都致力于探讨如何设计一个能自主进行操作的独立单元。对于一个由自主单元组成的网络，最优性和正确性的标准变得越来越复杂，特别是当控制系统也是分布式的时候。这就需要在独立单元的最优性和正确性之间将理论与实践相结合，以及将网络作为一个整体的最优性和正确性之间的理论与实践相结合。

挑战

如果可复用组件的质量及优化和验证的组合过程有开放的标准，复用设计就可以得到广泛的应用。这些标准覆盖了验证功能和性能的方法集合，包括模拟、硬件仿真、形式验证及不同层次的抽象。

挑战

有一个很清晰的需求，就是要有健全的基础作为基本原则，紧密结合设计开发、原型机制造和测试，进行有效的自我优化和自我验证的设计。挑战是在改善灵活性的同时适应性会提高，但也将优化和验证复杂化。

8.4 总结

基于下一代 SoC 和 ASoC，这是一个全新的研究领域。正如人们所看到的，晶体管密度的改进使得每平方厘米可以集成 10 亿只晶体管。如此巨大的计算潜力有一个核心限制：电能。这在体系结构领域打开了一个新方向——纳米计算。它与以往的针对超级计算机的工作截然不同，不存在竞争。这个领域的目标是在仅为现有功耗的百万分之一的级别下，研究高性能的算法和架构方法，同时释放芯片额外的功耗开销。

对于完美的操作，需要一种无线通信方式。这是另外一个非常大的挑战，特别是在微瓦级进行功率分配时。RF 是一种传统的方法，一些激光或红外线技术也许会提供新颖的解决方法。

另外，数字化传感器，甚至换能器，都存在一个不可避免的挑战——要将多个传感器集成到一个无缝的 SoC 中。

本章还规划了设计的前景，提出自我优化和自我验证组件，来处理由 ITRS 提出的设计挑战。描述了部署前后自我优化和自我验证的任务，同时讨论了可能的收益和挑战。在理论和实践上发展自我优化和自我验证都有助于达到研究目标——使得设计者能更快地完成更好的设计。

最好的设计要能够预测系统的复杂性，同时要有效地处理不可预测的因素。纵观本章系统复杂性包括很多因素：组件设计与电源、设计工具、验证与测试、安全性等。通过平衡这些问题来定义出有效的设计。

这里讨论的所有 ASoC 组件最终要整合到一个硅片上，但是有多种不同的组合方式。每一种拥有自己系统需求的组合，都必须通过各自独立的组件来优化。设计者在自我优化和自我验证方法的帮助下，已不再需要关注某个组件而是关注最终的系统。最终他们成为了系统设计师。

附录　处理器评估工具

假定给出了配置复杂的处理器结构，在没有仿真工具或预测工具帮助下，预测其性能或芯片面积总是不太可能的。

SimpleScalar 工具集适用于指令处理器的设计空间探索，较新版已能支持四种处理器架构：Alpha、ARM、PISA（MIPS 的变种）和 x86。

SimplrScalar 用户选择 Web 界面如图 A.1 所示，图 A.2、图 A.3 分别给出了不同的一级缓存（L1 cache）和不同的传输后备缓冲器（Translation Lookaside Buffer，TLB）配置情形下的仿真结果。

○ PISA + math + sim-cache
⊙ ARM + fmath + sim-bpred + not-taken
○ ARM + fmath + sim-bpred + bimod with large table size
○ x86 + llong + sim-outorder + 2 i-ALU + 2 f-ALU
○ x86 + llong + sim-outorder + 6 i-ALU + 6 f-ALU

Option	*x*-axis	Series
L1 cache	○	⊙
L2 cache	○	○
TLB	⊙	○

Option	*y*-axis
sim_IPC	⊙
Area	⊙

图 A.1　用户选择 Web 界面

该 SimpleScale Web 界面为用户提供了以下主要功能：

· 启用不同的 ISA，如 PISA、ARM 和 x86。
· 启用不同的基准测试程序，如 match. c、fmatch. c 和 llong. c。
· 针对各种处理器信息启用不同的 SimpleScalar 模拟器。
· 向 Web 用户浏览器所产生的图提供动态和实时更新。

如果想从 SimpleScalar 的 Web 界面产生一个图，首先需要选择架构；然后根据需要的仿真类型，选择相应的如一级缓存和 TLB 等信息选项；最后，可以选择模拟 IPC 值或区域信息。

如图 A.2 所示，x 轴表示不同的 TLB 的值，y 轴表示不同的仿真结果 IPC 值。如图 A.3 所示，图中每一点表示不同一级缓存配置下仿真结果 IPC 值。

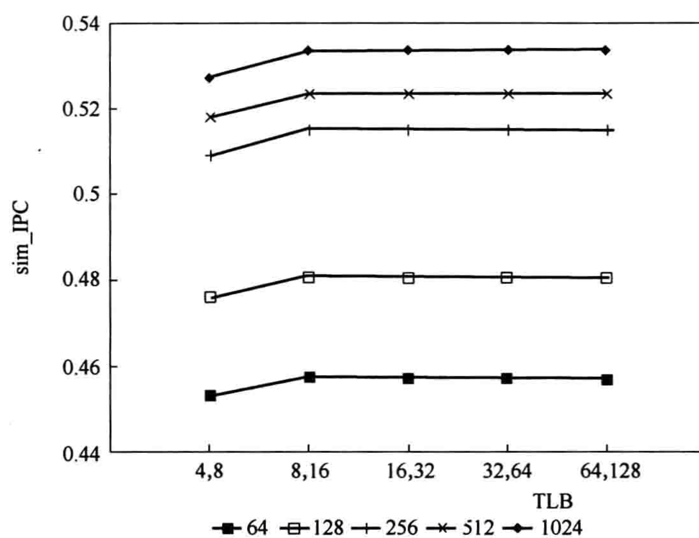


图 A.2 IPC 随 TLB 变化图

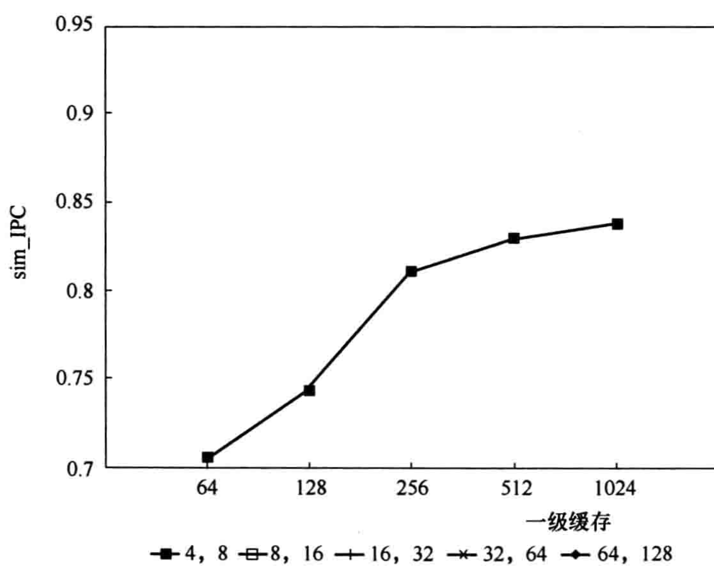


图 A.3 IPC 随一级缓存变化图

参考文献

[1] S. Abraham and K. Padmanabhan, "Performance of direct binary *n*-cube networks for multiprocessors," *IEEE Transactions on Computers*, 38(7):1000–1111, 1989.

[2] Actel, Axcelerator Family FPGAs, v2.8, 2009.

[3] Actel, IGLOO Handbook, v1.2, 2009.

[4] Actel, ProASIC Plus Family Flash FPGAs, v3.5, 2004.

[5] Actel, ProASIC3 Handbook, v1.4, 2009.

[6] F. Agakov et al., "Using machine learning to focus iterative optimization," *Proceedings of the International Symposium on Code Generation and Optimization*, IEEE, 2006, pp. 295–305.

[7] A. Agarwal, Analysis of Cache Performance of Operating Systems and Multiprogramming, PhD thesis, Computer Systems Laboratory, Stanford University, published as CSL-TR-87-332, 1987.

[8] A. Agarwal, "Limits on interconnection network performance," *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, 1991.

[9] K. Ajo, A. Okamura and M. Motomura, "Wrapper-based bus implementation techniques for performance improvement and cost reduction," *IEEE Journal of Solid-State Circuits*, 39(5):804–817, 2004.

[10] Altera, Avalon Interface Specifications, Version 1.2, 2009.

[11] Altera, Nios embedded processor, http://www.altera.com/products/ip/processors/nios/nio-index.html, 2010.

[12] Altera, Nios II Processor Reference Handbook Ver. 9.1, 2009.

[13] Altera, Nios II Performance Benchmarks, 2010.

[14] Altera, Stratix II Device Handbook, SII5V1-4.4, 2009.

[15] Altera, Stratix III Device Handbook, Version 2.0, 2010.

[16] Altera, Stratix IV Device Handbook, Version 4.2, 2010.

[17] H. Amano, "Japanese dynamically reconfigurable processors," *Proceedings of ERSA*, 2009, pp. 19–28.

[18] AMD, AMD Geode Brochure, 2005.

[19] ARC, ARC 600, Configurable 32-bit CPU core Description, 2005.

[20] ARM, ARM 1020E, Technical Reference Manual, rev. r1p7, 2003.

[21] ARM, AMBA Bus Standard Specifications, 2010.

[22] ARM, AMBA Specification, Rev 2.0, ARM-IHI-0011A.

[23] ARM, ARM VFP11, Vector Floating-Point Coprocessor for ARM1136JF-S Processor r1p5, Technical Reference Manual, 2007.

[24] ARM, ARM1136J(F)-S Processor Specifications, 2010.

[25] J.M. Arnold, "The architecture and development flow of the S5 software configurable processor," *Journal of VLSI Signal Processing*, 47(1):3–14, 2007.

[26] Arteris, "A comparison of network-on-chip and busses," White Paper, 2005.

[27] K. Asanovic et al., The landscape of parallel computing research: A view from Berkeley, Technical Report No. UCB/EECS-2006-183, 2006.

[28] K. Atasu et al., "CHIPS: Custom hardware instruction processor synthesis," *IEEE Transactions on Computer-Aided Design*, 27(3):528–541, 2008.

[29] K. Atasu et al., "Optimizing instruction-set extensible processors under data bandwidth constraints," *Proceedings of Design, Automation and Test in Europe Conference*, IEEE, 2007, pp. 1–6.

[30] T. Austin, E. Larson and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, 35(2):59–67, 2002.

[31] B. Bacheldor, "Belgium hospital combines RFID, sensors to monitor heart patients," *RFID Journal*, March 6, 2007.

[32] B. Bailey, G. Martin and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System-Level Methodology*, Morgan Kaufmann, 2007.

[33] J.E. Barth et al., "Embedded DRAM design and architecture for the IBM 0.11-μm ASIC offering," *IBM Journal of Research and Developmental*, 46(6):675–689, 2002.

[34] S. Baskiyar and N. Meghanathan, "A survey of contemporary real-time operating systems," *Informatica*, 29:233–240, 2005.

[35] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann and J. Luka, "Dynamic and partial FPGA exploitation," *Proceedings of the IEEE*, 95(2):438–452, 2007.

[36] T. Becker, W. Luk and P.Y.K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 35–44.

[37] T. Becker, W. Luk and P.Y.K. Cheung, "Parametric design for reconfigurable software-defined radio," *Reconfigurable Computing: Architectures, Tools and Applications*, LNCS 5453, J. Becker et al. (eds.), Springer, 2009.

[38] T. Becker, W. Luk and P.Y.K. Cheung, "Energy-aware optimisation for run-time reconfiguration," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 55–62.

[39] P. Beckett and S. Goldstein, "Why area might reduce power in nanoscale CMOS," *IEEE International Symposium on Circuits and Systems*, 3:2329–2332, 2005.

[40] L. Benini and G. De Micheli, "Networks on chips: A new SOC paradigm," *IEEE Computer*, 35(1):70–78, 2002.

[41] D.P. Bhandarkar, "Analysis of memory interference in multiprocessors," *IEEE Transactions on Computers*, C-24(9):897–908, 1975.

[42] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, (2nd ed.), Kluwer, 1997.

[43] J. Bicarregui, C.A.R. Hoare and J.C.P. Woodcock, "The verified software repository: A step towards the verifying compiler," *Formal Aspects of Computing*, 18(2):143–151, 2006.

[44] M. Birnbaum and H. Sachs, "How VSIA answers the SOC dilemma," *IEEE Computer*, 32(6):42–50, 1999.

[45] P. Biswas et al., "ISEGEN: Generation of high-quality instruction set extensions by iterative improvement," *Proceedings of DATE*, 2005, pp. 1246–1251.

[46] P. Boehm and T. Melham, Design and verification of on-chip communication protocols, Oxford University Computing Laboratory Research Report, RR-08-05, 2008.

[47] K. Bonsor, "How power paper will work," *How Stuff Works*, 12 January 2001.

[48] M. Borgatti et al., "A reconfigurable system featuring dynamically extensible embedded microprocessor, FPGA, and customizable I/O," *IEEE Journal of Solid-State Circuits*, 38(3):521–529, 2003.

[49] J.A. Bower et al., "Dynamic clock-frequencies for FPGAs," *Microprocessors and Microsystems*, 30(6):388–397, 2006.

[50] P. Brisk, A. Kaplan and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," *Proceedings of DAC*, 2004, pp. 395–400.

[51] D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Architecture News*, 25(3):13–25, 1997.

[52] M. Butts, A.M. Jones and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2007, pp. 55–64.

[53] Cadence Design Systems Inc, Palladium Datasheet, 2004.

[54] CEVA, Ceva X-1620 Product Note, 2005.

[55] K. Chen et al., "Predicting CMOS speed with gate oxide and voltage scaling and interconnect loading effects," *IEEE Transactions of the Electron Devices*, 44(11):1951–1957, 1997.

[56] D. Chen, J. Cong and P. Pan, "FPGA design automation: A survey," *Foundations and Trends in in Electronic Design Automation*, 1(3):139–169, 2006.

[57] C.T. Chow, L.S.M. Tsui, P.H.W. Leong, W. Luk and S. Wilton, "Dynamic voltage scaling for commercial FPGA," *Proceedings of the IEEE International Conference on Field-Programmable Technology*, National University of Singapore, 2005, pp. 173–180.

[58] S. Ciricescu et al., "The reconfigurable streaming vector processor (RSVP)," IEEE/ACM International Symposium on Microarchitecture MICRO-36, pp. 141–150, 2003.

[59] ClearSpeed, ClearSpeed CSX600 Datasheet, 2006.

[60] K. Compton and S. Hauck, "Totem: Custom reconfigurable array generation," *Proceedings of the Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2001, pp. 111–119.

[61] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, 34(2):171–210, 2002.

[62] G.A. Constantinides, "Word-length optimization for differentiable nonlinear systems," *ACM Transactions on Design Automation of Electronic Systems*, 11(1): 26–43, 2006.

[63] B.W. Cook, S. Lanzisera and K.S.J. Pister, "SoC issues for RF Smart Dust," *Proceedings of the IEEE*, 94(6):1177–1196, 2006.

[64] CrossBow Technologies, Xfabric Core Connectivity Junction, Preliminary Product Specification, 2004.

[65] R. Etienne-Cummings, P. Pouliquen and M.A. Lewis, "A vision chip for color segmentation and pattern matching," *EURASIP Journal on Applied Signal Processing*, 2003(7):703–712, 2003.

[66] U. Cummings, "PivotPoint: Clockless crossbar switch for high-performance embedded systems," *IEEE Micro*, 24(2):48–59, 2004.

[67] Cymbet, The POWER FAB (Thin Film Lithium Ion Cell) Battery System, 2007.

[68] Cypress semiconductor, CY8C41123 and CY8C41223 Linear Power PSoC Devices, 2005.

[69] J. Daemen and V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard*, Springer-Verlag, 2002.

[70] W.J. Dally, "Performance analysis of *k*-ary *n*-cube interconnection networks," *IEEE Transactions on Computers*, 39(6):775–785, 1990.

[71] W.J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," *Proceedings of the Design Automation Conference*, 2001.

[72] W.J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.

[73] S. Das et al., "A self-tuning DVS processor using delay-error detection and correction," *IEEE Journal of Solid-State Circuits*, 41(4):792–804, 2006.

[74] K. DeHaven, "Extensible processing platform ideal solution for a wide range of embedded systems," Xilinx White Paper WP369 (v1.0), 2010.

[75] J.A. DeRosa and H.M. Levy, "An evaluation of branch architectures," *Proceedings of the 14th Annual Symposium on Computer Architecture*, ACM, 10–16, 1987.

[76] A.S. Dhodapkarm and J.E. Smith, "Tuning adaptive microarchitectures," *International Journal of Embedded Systems*, 2(1/2):39–50, 2006.

[77] R. Dimond, O. Mencer and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proceedings—Computers and Digital Techniques*, 153(3):173–180, 2006.

[78] J. Duato, S. Yalamanchili and L. Ni, *Interconnection Networks*, Morgan Kaufmann, 2003.

[79] S. Dutta, "Architecture and implementation of multiprocessor SOCs for advanced set-top boxes and digital TV systems," *Proceedings of the 16th Symposium on Integrated Circuits and System Design*, 2003, pp. 145–146.

[80] C. Ebeling et al., "Implementing an OFDM receiver on the RaPiD reconfigurabl architecture," *IEEE Transactions on Computers*, 53(11):1436–1448, 2004.

[81] E. El-Araby, I. Gonzalez and T. El-Ghazawi, "Exploiting partial runtime recon figuration for high-performance reconfigurable computing," *ACM Transaction on Reconfigurable Technology and Systems*, 1(4):21, 2009.

[82] Elixent Corporation, DFA 1000 Accelerator Datasheet, 2003.

[83] Embedded Access, MQX RTOS Product Description, 2010.

[84] Fairchild Semiconductor, Two Input NAND Gate Layout, 1966.

[85] A. Fang et al., "Integrated hybrid silicon evanescent racetrack laser and photo detector," *12th OptoElectronics and Communications Conference*, 2007.

[86] A. Fauth, M. Freericks and A. Knoll, "Generation of hardware machine models from instruction set descriptions," *Proceedings of the IEEE Workshop VLSI Signal Processing*, IEEE, 242–250, 1993.

[87] A. Fauth, J. Van Praet and M. Freericks, "Describing instruction set processors using nML," *Proceedings of DATE*, IEEE, 503–507, March 1995.

[88] Federal Information Processing Standards publication 180-2, *Secure Hash Standard*, August 2002.

[89] A.K. Fidjeland and W. Luk, "Customising application-specific multiprocessor systems: A case study," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, IEEE, 239–244, 2005.

[90] A. Fidjeland, W. Luk and S. Muggleton, "A customisable multiprocessor for application-optimised inductive logic programming," *Proceedings of the Visions of Computer Science—BCS International Academic Conference*, September 2008, pp. 319–330.

[91] D. Fisch, A. Singh and G. Popov, "Z-RAM ultra-dense memory for 90nm and below," Hot Chips 18, August 2006.

[92] J.A. Fisher, "Very long instruction word architectures and the ELI-512," *Proceedings of the 10th Symposium on Computer Architecture*, ACM, 140–150, 1983.

[93] J.A. Fisher, P. Faraboschi and C. Young, *Embedded Computing*, Elsevier, 2005.

[94] J.A. Fisher, P. Faraboschi and C. Young, "Customizing processors: Lofty ambitions, stark realities," *Customizable Embedded Processors*, P. Ienne and R. Leupers (eds.), pp. 39–55, Morgan Kaufmann, 2007.

[95] D. Flynn, "AMBA: Enabling reusable on-chip designs," *IEEE Micro*, 17(4):20–27, 1997.

[96] M.J. Flynn, *Computer Architecture*, Jones and Bartlett, 1995.

[97] M.J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computing*, 21(9):948–960, 1972.

[98] M.J. Flynn and P. Hung, "Microprocessor design issues: Thoughts on the road ahead," *IEEE Micro*, 25(3):16–31, 2005.

[99] M.J. Flynn, P. Hung and K.W. Rudd, "Deep-submicron microprocessor design issues," *IEEE Micro*, 19(4):11–22, 1999.

[100] C.W. Fraser, D.R. Hanson and T.A. Proebsting, "Engineering a simple, efficient code-generator generator," *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992.

[101] Freescale Semiconductor, Freescale e600 Core Product Brief, Rev.0, 2004.

[102] Freescale Semiconductor, Freescale MPC8544E PowerQUICC III Integrated Processor, Hardware Specifications, Rev.2, 2009.

[103] Fujitsu, MB93555A Product Description, 2010.

[104] H. Fujiwara, *Logic Testing and Design for Testability*, MIT Press, 1985.

[105] S. Furber and J. Bainbridge, "Future trends in SoC interconnect," *Proceedings of the International Symposium on System-on-Chip*, 2005, pp. 183–186.

[106] Gaisler, Leon 4 Product Description, 2010.

[107] K. Gaj and P. Chodowiec, "Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays," *Proceedings of the RSA Security Conference*, 2001, pp. 84–99.

[108] A. Gerstlauer et al., "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design*, 28(10):1517–1530, 2009.

[109] S.K. Ghandi, *VLSI Fabrication Principles*, (2nd ed.), Morgan Kaufmann Publishers, 1994.

[110] D. Goodwin and D. Petkow, "Automatic generation of application specific processors," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003, pp. 137–147.

[111] H.H. Goode and R.E. Machol, *System Engineering—An Introduction to the Design of Large-Scale Systems*, McGraw-Hill, 1957.

[112] P. Guerrier and A. Grenier, "A generic architecture for on-chip packet-switched interconnections," *Proceedings of the IEEE Design Automation and Test in Europe*, IEEE, 250–256, 2000.

[113] I.J. Haikala, Program behavior in memory hierarchies, PhD thesis (Technical Report A-1986-2), University of Helsinki, 1986.

[114] A. Halambi and P. Grun, "Expression: A language for architecture exploration through compiler/simulator retargetability," *Proceedings of DATE*, March 1999, pp. 485–490.

[115] J. Hayter, *Probability and Statistics for Engineers and Scientists*, Duxbury Press, 2006.

[116] J. Heape and N. Stollon, "Embedded logic analyzer speeds SoPC design," *Chip Design Magazine*, August/September 2004.

[117] H. Hedberg, T. Lenart and H. Svensson, "A complete MP3 decoder on a chip," *Proceedings of the IEEE International Conference on Microelectronic Systems Education*, 2005, pp. 103–104.

[118] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, (4th ed.), Morgan Kaufmann, 2006.

[119] M. Hohenauer and R. Leupers, *C Compilers for Asips: Automatic Compiler Generation with LISA*, Springer, 2009.

[120] A.B.T. Hopkins and K.D. McDonald-Maier, "A generic on-chip debugger for wireless sensor networks," *Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems*, IEEE, 338–342, 2006.

[121] F. Hutter et al., "Boosting verification by automatic tuning of decision procedures," *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, IEEE, 27–34, 2007.

[122] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[123] IBM, 128-Bit Processor Logic Bus—Architecture Specification, Version 4.4, SA-14-2538-02, 2001.

[124] IBM, CoreConnect Bus Architecture, https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture, 2010.

[125] IBM, Embedded DRAM Comparison Charts, IBM Microelectronics Presentation, December 2003.

[126] IBM, On-chip Peripheral Bus—Architecture Specification, Version 2.1, SA-14-2528-02, 2001.

[127] P. Ienne and R. Leupers (eds.), *Customizable Embedded Processors*, Morgan Kaufmann, 2007.

[128] K. Illgner et al., "Programmable DSP platform for digital still cameras,' *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 4:2235–2238, 1999.

[129] Infineon, TriCore2, Synthesizable Processor Core, 2010.

[130] InSpeed, InSpeed SOC320, Emulex Overview, 2010.

[131] Intel, Intel IOP333 I/O Processor Datasheet, July 2005.

[132] Intel, Intel PXA27x Overview, 2010.

[133] ITRS, International Technology Roadmap for Semiconductors, 2009.

[134] ITRS, ITRS Roadmap Summary, 2006.

[135] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, 1991.

[136] D. Johnson, Handbook of Optical through the Air Communications, Imagineering E-Zine, 2008.

[137] J.R. Jump and S. Lakshmanamurthy, "NETSIM: A general-purpose interconnection network simulator," *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, H.D. Schwetman et al. (eds.), pp. 121–125, Society for Computer Simulation International, 1993.

[138] P.H.J. Kelly et al., "THEMIS: Component dependence metadata in adaptive parallel applications," *Parallel Processing Letters*, 11(4):455–470, 2001.

[139] J.O. Kephart and D.M. Chess, "The vision of autonomic computing," *IEEE Computer*, 36(1):41–50, 2003.

[140] D. Keymeulen et al., "Self-adaptive system based on field programmable gate array for extreme temperature electronics," *Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems*, IEEE, 296–300, 2006.

[141] M. Kistler, M. Perrone and F. Petrini, "Cell multiprocessor communication network: Built for speed,' *IEEE Micro*, 26(3):10–23, 2006.

[142] L. Kleinrock, *Queueing Systems: Theory, Vol. 1, Theory*, John Wiley and Sons, 1975.

[143] F. Kobayashi et al., "Hardware technology for Hitachi M-880 processor group," *Proceedings of the Electronic Components and Technologies Conference*, 693–703, 1991.

[144] T. Komuro, S. Kagami and M. Ishikawa, "A dynamically reconfigurable simd processor for a vision chip," *IEEE Journal of Solid-State Circuits*, 39(1):265–268, 2004.

[145] C. Kruskal and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Transactions on Computers*, C-32(12):1091–1098, 1983.

[146] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[147] K. Kutaragi et al., "A microprocessor with a 128 bit CPU, 10 floating-point MACs, 4 floating-point dividers, and an MPEG2 decoder," *IEEE International Solid-State Circuits Conference*, IEEE, 256–257, 1999.

[148] S.K. Lam and T. Srikanthan, "Rapid design of area-efficient custom instructions for reconfigurable embedded processing," *Journal of Systems Architecture*, 55(1):1–14, 2009.

[149] Lattice Semiconductor, *Lattice XP2 Family Handbook*, HB1004 Version 02.5, 2010.

[150] D. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, 24(12):1145–1154, 1975.

[151] E.A. Lee, "Embedded software," *Advances in Computers*, 56:56–97, 2002.

[152] F. Lee and A. Dolgoeorodov, "Implementation of H.264 encoding algorithms on a software-configurable processor," *Proc. GSPx*, 2005.

[153] D. Lee et al., "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design*, 25(10):1990–2000, 2006.

[154] J.K.F. Lee and A.J. Smith, "Analysis of branch prediction strategies and branch target buffer design," *IEEE Computer*, 17(1):6–22, 1984.

[155] O. Lehtoranta et al., "A parallel MPEG-4 encoder for FPGA based multiprocessor SOC," *Proceedings of the IEEE ISCAS*, 2005.

[156] S. Leibson, "NOC, NOC, NOCing on heaven's door: Beyond MPSOCs," *Electronics Design, Strategy, News*, 8 December 2005.

[157] G. Lemieux and D. Lewis, *Design of Interconnect Networks for Programmable Logic*, Kluwer, 2004.

[158] V. Liguori and K. Wong, "Designing a real-time *HDTV* 1080p baseline H.264/AVC encoder core," *Proceedings of DesignCon*, 2006.

[159] W. Luk et al., "A high-level compilation toolchain for heterogeneous systems,"

Proceedings of the IEEE International SOC Conference, 2009, pp. 9–18.

[160] D. Lyonnard, S. Yoo, A. Baghdadi and A.A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," *Proc. Design Automation Conference*, 518–523, IEEE, May 2001.

[161] P. Lysaght and D. Levi, "Of gates and wires," *International Parallel and Distributed Processing Symposium*, 2004.

[162] P. Machanick, "SMP-SOC is the answer you get if you ask the right questions," *Proceedings of SAICSIT*, SAICSIT, 12–21, 2006.

[163] T. Makimoto, "The hot decade of field programmable technologies," *Proceedings of the IEEE International Conference on Field-Programmable Technology*, IEEE, 3–6, 2002.

[164] G. Martin and H. Chang (eds.), *Winning the SoC Revolution*, Kluwer, 2003.

[165] M.M. Mbaye, N. Blanger, Y. Savaria and S. Pierre, "A novel application-specific instruction-set processor design approach for video processing acceleration," *Journal of VLSI Signal Processing Systems*, 47(3):297–315, 2007.

[166] G. McFarland, CMOS Technology Scaling and its Impact on Cache Delay, PhD thesis, Stanford University, 1997.

[167] J. McGregor, Interconnects target SoC design, Microprocessor Report, 2004.

[168] S. McKeever and W. Luk, "Provably-correct hardware compilation tools based on pass separation techniques," *Formal Aspects of Computing*, 18(2):120–142, 2006.

[169] B. McNamara, M. Ji and M. Leabman, "Implementing 802.16 SDR using a software-configurable processor," *Proceedings of GSPx*, 2005.

[170] C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[171] B. Mei et al., "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," *Field-Programmable Logic and Applications*, LNCS 2778, P.Y.K. Cheung, G.A. Constantinides and J.T. de Sousa (eds.), Springer, 2003.

[172] A. Mello, L. Moller, N. Calazans and F. Moraes, "MultiNoC: A multiprocessing system enabled by a network on chip," *Proceedings of Design, Automation and Test in Europe*, IEEE, 234–239, 2005.

[173] S. Meninger et al., "Vibration-to-electric energy conversion," *IEEE Transactions of the VLSI Systems*, 9(1):64–76, 2001.

[174] Mentor Graphics, Atsana Semiconductor J2211 Product Description, 2010.

[175] Mentor Graphics, Nucleus Operating System, 2010.

[176] Microprocessor Report, Matsushita Integrated Platform, 2005.

[177] Microprocessor Report, MicroBlaze Can Float, 5/17/05-02, 2005.

[178] Microprocessor Report, XAP3 Takes the Stage, 6/13/05-01, 2005.

[179] P. Mishra and N. Dutt (eds.), *Processor Description Languages, Applications and Methodologies*, Morgan Kaufmann, 2008.

[180] S. Mirzaei, A. Hosangadi and R. Kastner, "FPGA implementation of high speed FIR filters using add and shift method," *Proceedings of ICCD*, 2006, pp. 308–313.

[181] A. Molnar et al., "An ultra low power 900 MHz RF transceiver for wireless sensor output," *Proceedings of the Custom Integrated Circuits Conference*, IEEE, 2004, pp. 401–404.

[182] A.C. Murray, R.V. Bennett, B. Franke and N. Topham, "Code transformation and instruction set extension," *ACM Transactions on Embedded Computing*, 8(4), Article 26, 2009.

[183] MIPS, MIPS 74K Core Product Description, 2010.

[184] NetSilicon, NET+Works for NET+ARM, Hardware Reference Guide, 2000.

[185] NetSilicon, NetSilicon NS9775 Datasheet, Rev. C, January 2005.

[186] NXP, LH7A404, 32-Bit System-on-Chip, Preliminary data sheet, July 2007.

[187] M. Oka and M. Suzuoki, "Designing and programming the Emotion Engine," *IEEE Micro*, 19(6):20–28, 1999.

[188] Open Core Protocol International Partners, Open Core Protocol Specification 1.0, OCP-IP Association, Document Version 002, 2001.

[189] OpenCores, Wishbone B4, 2010.

[190] OpenCores, OpenRISC, 2010.

[191] I. Page and W. Luk, "Compiling occam into FPGAs," *FPGAs*, W. Moore and W. Luk (eds.), pp. 271–283, Abingdon EE&CS books, 1991.

[192] R. Paschotta, Encyclopedia of Laser Physics and Technology, RP Photonics, 2010.

[193] S. Pasricha and N. Dutt, *On-Chip Communication Architectures*, Morgan Kaufmann, 2008.

[194] J.H. Patel, "Performance of processor–memory interconnections for multiprocessors," *IEEE Transactions on Computers*, 30(10):771–780, 1981.

[195] L.D. Partain, *Solar Cells and Their Applications*, Wiley, 2004.

[196] O. Pell, "Verification *of FPGA* layout generators in higher order logic," *Journal of Automated Reasoning*, 37(1–2):117–152, 2006.

[197] O. Pell and W. Luk, "Instance-specific design," *Reconfigurable Computing*, S. Hauck and A. DeHon (eds.), pp. 455–474, Morgan Kaufmann, 2008.

[198] P. Pelgrims, T. Tierens and D. Driessens, Evaluation Report OCIDEC-Case, De Nayer Instituut., 2003.

[199] Philips, Nexperia PNX1700 Connected Media Processor, 2007.

[200] M. Porrmann, U. Witkowski and U. Rueckert, "Implementation of self-organizing feature maps in reconfigurable hardware," *FPGA Implementations of Neural Networks*, A.R. Omondi and J.C. Rajapakse (eds.), 247–269, Springer, 2006.

[201] E.J. Prinz et al., "Sonos: An embedded 90nm SONOS flash EEPROM utilizing hot electron injection programming and 2-sided hot hole injection erase," *IEDM Conference Record*, 2002.

[202] S. Przybylski, M. Horowitz and J. Hennessy, "Characteristics of performance optimal multi-level cache hierarchies," *Proceedings of the 16th Symposium on Computer Architecture*, ACM, 114–121, June 1989.

[203] V.L. Pushparaj et al., "Flexible energy storage devices based on nanocomposite paper," *Proceedings of the National Academy of the USA*, 104(34):13574–13577, 2007.

[204] C.V. Ravi, "On the bandwidth and interference in interleaved memory systems," *IEEE Transactions on Computers*, 21(8):899–901, 1972.

[205] RFID Journal, http://www.rfidjournal.com.

[206] S. Roundy et al., "Power sources for wireless sensor networks," *Proceedings of the 1st European Workshop on Wireless Sensor Networks*, 2004, pp. 1–17.

[207] C. Rowen and S. Leibson, *Engineering the Complex SoC*, Prentice Hall, 2004.

[208] R.M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, 21(1):63–72, 1978.

[209] S. Sane, "The aerodynamics of insect flight," *The Journal of Experimental Biology*, 206:4191–4208, 2003.

[210] J. Schmaltz and D. Borrione, "A generic network on chip model," *Theorem Proving in Higher Order Logics*, LNCS 3603, J. Hurd and T. Melham (eds.), pp. 310–325, Springer, 2005.

[211] P. Sedcole et al., "Run-time integration of reconfigurable video processing systems," *IEEE Transactions on VLSI Systems*, 15(9):1003–1016, 2007.

[212] P. Sedcole and P.Y.K. Cheung, "Parametric yield modelling and simulations of FPGA circuits considering within-die delay variations," *ACM Transactions on Reconfigurable Technology and Systems*, 1(2), Article 10, 2008.

[213] S. Seng, W. Luk and P.Y.K. Cheung, "Run-time adaptable flexible instruction processors," *Field Programmable Logic and Applications*, LNCS 2438, M. Glesner, P. Zipf and M. Renovell (eds.), pp. 545–555, 2002.

[214] R.A. Shafik, B.H. Al-Hashimi and K. Chakrabarty, "Soft error-aware design optimization of low power and time-constrained embedded systems," *Proceedings of DATE*, IEEE, 1462–1467, 2010.

[215] L. Shannon and P. Chow, "SIMPPL: An adaptable SoC framework using a programmable controller IP interface to facilitate design reuse," *IEEE Transactions on VLSI Systems*, 15(4):377–390, 2007.

[216] N. Shirazi, W. Luk and P.Y.K. Cheung, "Run-time management of dynamically reconfigurable designs," *Field-Programmable Logic and Applications*, LNCS 1482, R.W. Hartenstein and A. Keevallik (eds.), pp. 59–68, Springer, 1998.

[217] M. Shirvaikar and L. Estevez, "Digital camera design with JPEG, MPEG4, MP3 and 802.11 features," *Embedded Systems Conference*, 2006.

[218] M.L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*, Wiley, 2001.

[219] SiliconBlue, iCE65 Ultra Low-Power Mobile FPFA Family, 2.1.1, 2010.

[220] Silicon Hive, Avispa block accelerator, Product Brief, 2003.

[221] A.J. Smith, "Cache evaluation and the impact of workload choice," *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 64–73, ACM, 1985.

[222] J.E. Smith, "A study of branch prediction strategies," *Proceedings of the*

Symposium on Computer Architecture, pp. 135–148, ACM, 1981.

[223] A.J. Smith, "Cache memories," *ACM Computing Surveys*, 14(3):473–530, 1982.

[224] A.J. Smith, "Cache evaluation and the impact of workload choice," *Proceedings of the International Symposium on Computer Architecture*, pp. 64–73, ACM, 1985.

[225] Sonics Inc, Sonics μNetwork Technical Overview, Document Revision 1, 2002.

[226] B. Stackhouse el al. "A 65 nm 2-billion-transistor quad-core Itanium processor," *IEEE Journal of Solid-State Circuits*, 44(1):18–31, 2009.

[227] T. Starnes, Programmable Microcomponent Forecast through 2006, Gartner Market Statistics, 2003.

[228] H.S. Stone, *High-Performance Computer Architecture*, (2nd ed.), AddisonWesley, 1990.

[229] W.D. Strecker, Analysis of the Instruction Execution Rate in Certain Computer Systems, PhD thesis, Carnegie-Mellon University, 1970.

[230] Stretch, "The S6000 family of processors," Architecture White Paper, 2009.

[231] H.E. Styles and W. Luk, "Exploiting program branch probabilities in hardware compilation," *IEEE Transactions on Computers*, 53(1):1408–1419, 2004.

[232] H.E. Styles and W. Luk, "Compilation and management of phase-optimized reconfigurable systems," *Proceedings of the International Conference on Field-Programmable Logic and Applications*, IEEE, 311–316, 2005.

[233] T. Sugizaki et al., "Novel multi-bit sonos type flash memory using a high-k charge trapping layer," *IEEE Symposium on VLSI Technology, Digest of Technical Papers*, IEEE, 27–28, June 2003.

[234] Sun, "Java Card 3 Platform," White Paper, 2008.

[235] Sun, OpenSPARC T1 FPGA Implementation, Release 1.6 Update, 2008.

[236] K.W. Susanto, "An integrated formal approach for system on chip," *Proceedings of the International Workshop in IP Based Design*, 119–123, October 2002.

[237] M. Suzuoki et al., "A microprocessor with a 128-bit CPU, ten floating-point MAC's, four floating-point dividers, and an MPEG-2 decoder," *IEEE Journal of Solid-State Circuits*, 34(11):1608–1618, 1999.

[238] D. Sykes et al., "Plan-directed architectural change for autonomous systems," *Proceedings of the International Workshop on Specification and Verification of Component-Based Systems*, 2007, pp. 15–21.

[239] Target Compiler Technologies, The nML Processor Description Language, 2002.

[240] Tensilica, Tensilica Instruction Extension (TIE) Language Reference Manual, 2006.

[241] R. Tessier et al., "A reconfigurable, power-efficient adaptive Viterbi decoder," *IEEE Transactions on VLSI Systems*, 13(4):484–488, 2005.

[242] J.E. Thornton, *Design of a Computer: The Control Data 6600*, Scott, Foresman and Co., 1970.

[243] Texas Instruments, TMS320C6713B, Floating point digital signal processor Datasheet, Rev. B, 2006.

[244] T.J. Todman et al., "Reconfigurable computing: Architectures and design methods," *IEE Proceedings—Computers and Digital Techniques*, 152(2):193–207, 2005.

[245] T. Todman, J.G. de, F. Coutinho and W. Luk, "Customisable hardware compilation," *The Journal of Supercomputing*, 32(2):119–137, 2005.

[246] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[247] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.

[248] J. Villarreal, A. Park, W. Najjar and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010.

[249] Virtual Socket Interface Alliance, On-Chip Bus DWG, Virtual Component Interface (VCI) Specification Version 2, OCB 2 2.0, 2001.

[250] W.J. Watson, "The TI ASC: A highly modular and flexible super computer architecture," *Proceedings of the AFIPS*, 41(1):221–228, 1972.

[251] B. Wen and K. Boahen, "Active bidirectional coupling in a cochlear chip," *Advances in Neural Information Processing Systems 17*, B. Sholkopf and Y. Weiss (eds.), MIT Press, 2006.

[252] N. Whitehead, M. Abadim and G. Necula, "By reason and authority: A system for authorization of proof-carrying code," *Proceedings of the IEEE Computer Security Foundations Workshop*, IEEE, 236–250, 2004.

[253] S.J.E. Wilton et al., "A synthesizable datapath-oriented embedded FPGA fabric for silicon debug applications," *ACM Transactions on Reconfigurable Technology and Systems*, 1(1), Article 7, 2008.

[254] Wind River, Wind River VxWorks, http://www.windriver.com/products/vxworks, 2010.

[255] R. Wood, "Fly, robot fly," *IEEE Spectrum*, 45(3):21–25, 2008.

[256] C.-L. Wu and T.-Y. Feng, "On a class of multistage interconnection networks," *IEEE Transactions on Computers*, 29(8):694–702, 1980.

[257] Xelerated, Xelerator X10q Network Processors, Product Brief, 2004.

[258] Xilinx, MicroBlaze Processor Reference Guide, EDK 11.4, 2009.

[259] Xilinx, Microblaze Processor Reference Guide, 2004.

[260] Xilinx, MicroBlaze Soft Processor Core, http://www.xilinx.com/tools/microblaze.htm, 2010.

[261] Xilinx, PowerPC 405 Processor Block Reference Guide, 2003.

[262] Xilinx, Virtex II Datasheet, 2004.

[263] Xilinx, Virtex 4 FPGA User Guide, v2.6, 2008.

[264] Xilinx, Virtex 5 FPGA User Guide, v5.3, 2010.

[265] Xilinx, Virtex-6 Family Overview, v2.2, 2010.

[266] E.M. Yeatman, "Advances in power sources for wireless sensor nodes," *Proceedings of the 1st International Workshop on Body Sensor Networks*, 2004.

[267] T.-Y. Yeh and Y.N. Patt, "Alternative implementations of two-level adaptive branch prediction," *Proceedings of the International Symposium on Computer Architecture*, ACM, 124–134, May 1992.

[268] T.-Y. Yeh and Y.N. Patt, "Two-level adaptive training branch prediction," *Proceedings of the International Symposium on Microarchitecture*, IEEE, 51–61, November 1991.

[269] P. Yianancouras, J.G. Steffan and J. Rose, "Exploration and customization of FPGA-based soft processors," *IEEE Transactions on Computer-Aided Design*, 26(2):266–277, 2007.

[270] A.C. Yu, Improvement of Video Coding Efficiency for Multimedia Processing, PhD thesis, Stanford University, 2002.

[271] B. Zhai et al., "A 2.60pJ/inst subthreshold sensor processor for optimal energy efficiency," *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*, IEEE, 2006, pp. 154–155.

[272] Z. Zhang et al., "AutoPilot: A platform-based ESL synthesis system," *HighLevel Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec (eds.), Springer Publishers, 2008.

[273] J. Zufferey and D. Floreano, "Toward 30-gram autonomous indoor aircraft: Vision-based obstacle avoidance and altitude control," *Proceedings of the IEEE International Conference on Robotics and Automation*, 2005, pp. 2594–2599.

国际视野　科技前沿

国际信息工程先进技术译丛

《计算机系统设计：片上系统》
《无线传感器网络——原理、设计和应用》
《IPv6部署和管理》
《虚拟网络——下一代互联网的多元化方法》
《下一代融合网络理论与实践》
《认知视角下的无线传感器网络》
《移动云计算：无线、移动及社交网络中分布式资源的开发利用》
《Android系统安全与攻防》
《内容分发网络》
《计算机网络仿真OPNET实用指南》
《移动无线信道》（原书第2版）
《LTE-Advanced：面向IMT-Advanced的3GPP解决方案》
《声学成像技术及工程应用》
《认知无线电通信与组网：原理与应用》
《LTE/SAE网络部署实用指南》
《网络性能分析原理与应用》
《云连接与嵌入式传感系统》
《IP地址管理原理与实践》
《自组织网络：GSM，UMTS和LTE的自规划、自优化和自愈合》
《实现吉比特传输的60GHz无线通信技术》
《LTE自组织网络（SON）：高效的网络管理自动化》
《UMTS中的LTE：向LTE-Advanced演进》（原书第2版）
《无线传感器及执行器网络》
《UMTS中的WCDMA－HSPA演进及LTE》（原书第5版）
《认知无线电网络》
《网络融合——服务、应用、传输和运营支撑》
《UMTS中的LTE:基于OFDMA和SC-FDMA的无线接入》
《高性能微处理器电路设计》
《吉规模集成电路互连工艺及设计》
《高级电子封装》（原书第2版）
《基于4G系统的移动服务技术》
《移动无线传感器网——技术、应用和发展方向》
《UMTS蜂窝系统的Q0S与Q0E管理》
《UMTS-HSDPA系统的TCP性能》
《基于射频工程的UMTS空中接口设计与网络运行》
《未来UMTS的体系结构与业务平台：全IP的3GCDMA网络》
《环境网络：支持下一代无线业务的多域协同网络》
《基于蜂窝系统的IMS—融合电信领域的V0IP演进》
《蜂窝网络高级规划与优化 2G/2.5G/3G/——向4G的演进》
《微电子技术原理、设计与应用》
《多电压CMOS电路设计》
《P2P系统及其应用》
《IPTV与网络视频：拓展广播电视的应用范围》



机械工业出版社微信公众号

WILEY

上架指导　计算机／计算机科学



ISBN 978-7-111-49813-1　定价：78.00元